IEEE MICRO

JUNE 1990

Chips, Systems, Software, and Applications

More Design Tips from the First Hot Chips Symposium

Special Feature: The Gmicro/300 Processor

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE COMPUTER SOCIETY

Cover image: Image Bank
Cover design: Design & Direction

IEEE MICRO

Volume 10 Number 3 (ISSN 0272-1732) June 1990

Published by the IEEE Computer Society

Departments

Features

Special Feature

IEEE Computer Society
PO Box 3014
Los Alamitos, CA 90720-1264
(714) 821–8380

Circulation: *IEEE Micro* (ISSN 0272-1732) is published bimonthly by the IEEE Computer Society, PO Box 3014, Los Alamitos, CA 90720-1264; IEEE Computer Society Headquarters, 1730 Massachusetts Ave., NW, Washington, DC 20036-1903; IEEE Headquarters, 345 East 47th St., New York, NY 10017. Annual subscription: $19 in addition to IEEE Computer Society or any other IEEE society member dues; $35 for members of other technical organizations. Back issues: $10 for members; $20 for nonmembers. This journal is also available in microfiche form.

Postmaster: Send address changes and undelivered copies to *IEEE Micro*, PO Box 3014, Los Alamitos, CA 90720-1264. Second-class postage is paid at New York, NY, and at additional mailing offices.

Editorial: Unless otherwise stated, bylined articles and descriptions of products and services reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society. Send editorial correspondence to *IEEE Micro*, PO Box 3014, Los Alamitos, CA 90720-1264. All submissions are subject to editing for style, clarity, and space considerations.

IEEE Micro

* Submit six copies of all articles and special-issue proposals to: Joe Hootman, EE Dept., University of North Dakota, PO Box 7165, Grand Forks, ND 58202; (701) 777-4331; Compmail+ j.hootman.

Letters

The Futurebus+ protocol stack and profiles

[In last December's Micro Standards column, Carl Warren discussed the IEEE 896 Futurebus standard. Harrison Beasley, the vice chair of the IEEE 896.2 committee, sent in some additional material, only part of which we can reproduce here due to page limitations. This material will further our understanding of the standard as well as expand and revise some of the descriptions Carl Warren provided.

Direct any questions, comments, or requests for additional information to LCDR Harrison Beasley, Vice Chair IEEE 896.2, 10408 Collingham Drive, Fairfax, VA 22032.—Ed.]

To the Editor:

The IEEE 896 project, originally conceived in the late 1970s and extensively engineered throughout the 80s, is about to become an implementation reality. The goals of this effort—an architecture, processor, and technology-independent backplane interconnect that provides a quantum leap over existing open buses—have been realized.

The project spawned several companion working groups due to the realization that developing independent specifications with a layered approach to end-to-end communications was much more useful than just another bus specification. With the impending completion of several of these standards, a method of combining the desired features of each into a cohesive whole is required. For Futurebus+, this method is known as an Application Environment Profile. An AEP, or profile, describes system functional requirements and points to existing standards, selecting and binding options where necessary for interoperability within those standards.

The Futurebus+ protocol stack (see figure on p. 92) is made up of the following separate IEEE standards: P1101.*x*, Mechanical Core Specification for Eurocard Form Factors and 2-mm Pitch Metric Connectors; P1194.1, Electrical Characteristics of Backplane Transceiver Logic Circuits; P1212, Control and Status Registers (CSR); 896.1 Futurebus+ Logical Layer; and 896.2 Futurebus+ Physical Layer and Profiles.

The P896.3, Futurebus+ Systems Configuration Standard, P1394 Serial Bus, and other standards may be invoked in a profile. The profile provides a cohesive thread through these documents ensuring interoperability across a multivendor support base. The following paragraphs describe layer by layer this stack, providing pertinent information at each layer.

Any system has mechanical constraints imposed on it. Numerous board-based systems use the Eurocard format for mechanical interoperability. The 896 working group selected 6U × 280 mm as the preferred profile A board size. Recognizing the applicability of Futurebus+ in multiple computing environments, the specification makes provision for other heights (9U), or other depths (annotated as A6-160, A9-220, ...). Military systems are expected to also use the Standard Electronic Module form factors. Any new profile desiring mechanical interchangeability with existing profiles would mandate a board size(s) and connector compatible with those previously specified. IEEE 1101.2 contains the specification for Eurocard form factors and connector selected by the Futurebus+ working group. Board spacing is also specified in this layer.

Electrical interface requirements are specified in 896.1 chapter 4 (technology-independent requirements) and 896.2 chapters 3 (power distribution) and 4 (requirements for boards using BTL drivers). Backplane Transceiver Logic (BTL) drivers, specified in IEEE 1194.1, provide incident wave switching, low capacitance, and controlled voltage swings, yielding an elegant solution to the "bus-driving problem." All profiles would adhere to the 896.1 and 896.2 chapter 3 material. Selection of different drivers would obviously require another specification.

Pinouts are specified in 896.2. A profile is free to make use of existing pinouts or define one of its own. However, the requirement of interoperability of products from multiple vendors mandate that Futurebus+ location, pinout, and connector footprint be consistent. Profile A also specifies that the upper signal connector on 6U and 9U boards

continued on p. 92

From the Editor-in-Chief

Hot Chips, Part 2

This issue of *IEEE Micro* contains the second half of the papers presented at last year's IEEE Computer Society's First Annual Hot Chips Symposium. We again thank Hot Chips chair Bob Stewart for his untiring efforts in working with each author to prepare their papers for publication in *Micro*.

In the first article David James recounts the continuing battle over the correct position of the high and low bytes of data in a register. This ongoing argument impacts the design of buses, a crucial decision for designers.

Edenfield and others present the second, and final, part of the discussion of the Motorola 68040. They cover the memory subsystem, the external bus, and design verification.

Sun Microsystems and Texas Instruments cooperated to develop a high-performance floating-point processor for the Sparc system. Merrick Darley and others discuss this new TI chip, which is capable of 5.5 million double-precision floating-point operations a second.

The final Hot Chips article features Mitch Alsup's detailed discussion of the Motorola 88000 design. Mitch talks about the 88100 processor and the 88200 memory management units from the system performance and software performance points of view.

A special feature in this issue is Kitahara and Satoh's article about Fujitsu's Gmicro/300, a high-performance processor based on the TRON architecture. This article, originally part of *Micro*'s annual Far East issue, gives you one more look at current Japanese technology.

Also with this issue we lose the services and expertise of four Editorial Board members. K.-E. Grosspietsch and Dante Del Corso served *Micro*'s best interests for four years. During their tenures they acted as co-guest editors for several issues, presenting the European perspective on research in microcomputers and related areas. Their most recent collaboration (December 1989 and one of our more popular issues) addressed the realization of neural networks.

Jay Kamdar and Varish Panigrahi, active members of the Editorial Board for a number of years, provided many ideas for the magazine and were excellent reviewers. The standards that these individuals enforced in the review process helped produce the high quality of the material you read in *Micro*.

On behalf of the readers and myself, I wish to thank these outstanding individuals for their contributions to *IEEE Micro*. I also wish them the best in their future endeavors.

Beginning in the next issue, we introduce a column that focuses on software research but also touches on hardware. Software Report consists of a series of reports from Japan written by Editorial Board Member David Kahaner. David recently accepted an assignment to establish communications between researchers in Japan and the US.

We believe you will enjoy these edited trip reports, get a glimpse of the current software research topics that are of interest in Japan, and perhaps establish beneficial contacts with your counterparts in Japan.

As always, we look forward to hearing from you. We value your comments.

In the mailbag

August 1989

"I would like a list of technical books that you produce. I would like to see your country very soon." R.A., Tehran, Iran (Your request for a catalog of books has been forwarded to the IEEE Computer Society Press.—J.H.)

October 1989

"I want *The NeXT Book*. Please let me know how to buy it." J.H.J., Seoul, Korea (I've forwarded your request to the book's publishers.—J.H.)

"I liked all [the] topics in *Micro* magazine." R.M.A., Mosul, Iraq

"I liked all [the] contents of *IEEE Micro*." A.A., Mosul, Iraq

"I would like to see a short abstract at the start of each article. I'd also like to see more about distributed memory machines." S.G., Newcastle-Upon-Tyne, U.K. (Short abstracts now appear on the first pages of each article in *IEEE Micro* and are going to become a standard part of all of the articles appearing in each IEEE Computer Society magazine.—J.H.]

"I liked 'The Micon System for Computer Design'—a big help for today's workstation design." G.C., VS-Villingen, West Germany

December 1989

"I liked 'An Analog VLSI Implementation of Hopfield's Neural Network.'" I.S., Ulsan, Korea

"I liked your feature article, 'VLSI Architectures for Neural Networks,' that gave me an in-depth knowledge with regards to neural computing. I disliked the redundancy on some of the parts of [the] other articles [in] the same issue. I would like to see a box which defines some new terms that are related in the articles like in *Computer* magazine." A.S.Y., Quezon City, The Philippines (Redundancy in an area as new as neural network realization is part of the process of starting a new field of research and development. The area needs to have some time to mature and to develop more literature.—J.H.)

"I would like to see more articles on RISC." K.S.B., Sacramento, CA (There are a great number of RISC articles in the two 1990 Hot Chips issues.—J.H.)

"I would like to see more articles about petroleum and related fields." A.R.K., Tehran, Iran (It is not likely that *Micro* will publish many, if any, articles in this area.—J.H.)

"I'm working on neural networks, and I will have a lecture on this material. I would like to see some literature in this field." M.A.Z., Shiraz, Iran

"I liked On the Edge and Micro Law." J.M.W., Geneva, Switzerland

"On the Edge is much needed in the Micro world." E.L., Nedlands, Australia

"I liked the issue on neural networks altogether! I would like to see more of the same." L.S.S., Stirling, Scotland

"I liked [all of the] neural networks [articles]." T.K.K., Shatin, Hong Kong

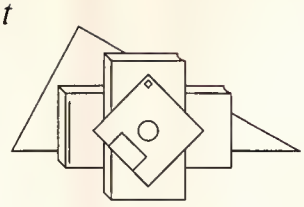
"I liked the author's request for more knowledge on fundamental concepts; also the statement on the top half of page 76: '...no sane physicist would claim to know what *space* or *time* means.'" K.A.R., Vasteras, Sweden (This refers to the short article, "Is There a Silicon Way to Intelligence?" by E.R. Caianiello.—J.H.)

February 1990

"I liked February 1990's On the Edge discussion of buses, or should I say I was interested? I think Carl Warren's technical points (e.g., grounding theory) are shaky. I would like to see innovative algorithms in pseudo code." T.J.S., Menlo Park, CA

"I would like to see IEEE publications distributed through an online subscription service." W.W.L., Milwaukee, WI (Several groups in the IEEE Computer Society are investigating the best ways for the CS to present material to its members. One method that is being considered is an online subscription service.—J.H.)

Department



Micro Review

Richard Mateosian
Hitachi America, Ltd.
(415) 244-7456
Fax (415) 583-4207

The book we've all been waiting for

For some time now, the two people most associated in the press with RISC computers have been collaborating on a book on computer architecture and design. It has finally appeared after a year of beta testing, and it is truly different from—and better than—its predecessors. Gordon Bell says in his foreword, "To advance computing, I urge publishers to withdraw the scores of books on this topic so a new breed of architect/engineer can quickly emerge." Publishers won't, of course, but if you're teaching a course in this area, you can make your own decision.

Computer Architecture—A Quantitative Approach, David A. Patterson and John L. Hennessy (Morgan Kaufmann, San Mateo, Calif., 1990, 758 pp., $54.95)

Dave Patterson coined the term reduced instruction-set computer (RISC) in connection with his work at the University of California at Berkeley. This work ultimately led to the Sparc architecture, now widely used in workstations. John Hennessy's work at Stanford University led to another RISC architecture used in computers made by MIPS Computer Systems and their licensees. Not surprisingly, their book marries the theoretical analysis of academe with the practical concerns of industry. This marriage is what engineering is about, but prior books on computer engineering have rarely achieved this synthesis. Furthermore, the authors write directly and clearly. As one of my colleagues put it, they wrote the book in English. (I don't claim to have read every word of this book, but I did notice that they misspelled *visible* on page 555.)

The subtitle of the book tells much of the story. The authors examine the issues involved in quantifying the costs and the performance deltas of architectural alternatives. They do so with examples and exercises that refer to current mainline architectures, like the IBM 370, Intel 80X86, and DEC VAX.

Each chapter ends with sections called Putting It All Together, Fallacies and Pitfalls, Concluding Remarks, and Historical Perspectives, followed by excellent sets of exercises. The fallacies and pitfalls section is extremely important in this field. Marketing hype and unfounded theorizing have combined to create a large body of myth, but even the most elementary pitfalls have cost many millions of dollars. For example, many semiconductor firms have launched projects with inadequate goals, simply from failing to heed the first pitfall listed in the book: ignoring the inexorable progress of hardware when planning a new machine.

The authors detail some of the many pitfalls surrounding performance measurement and comparison. Other pitfalls include the most well-known of all: too small an address space. The authors' RISC work has led them to strong opinions on the pitfalls of instruction-set design, but even the strongest complex instruction-set computer proponents will probably find little to disagree with.

The fallacies concerning implementation techniques and pipelining are a little arcane, but those on vector machines touch on the popular because of the commercial significance of performance comparisons. Chapter 9 contains a marketing-oriented fallacy concerning the computation of average seek times on a disk. Even chapter 10, which deals speculatively with future directions, contains interesting fallacies, again related to performance analysis.

While the analysis of pitfalls and fallacies is important, the most important thing about the book is that it teaches the techniques that designers in industry have been using all along. In contrast with other computer architecture books I have looked at (which contain material that seems quite foreign to me), this book contains precisely the subject matter of product-planning discussions among good computer architects in the industry.

In summary, this book is head and shoulders above anything else available. If you are teaching a class, you have no excuse for using any other textbook. If you just want to read about the subject, you can't find a better book.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 177 **Medium** 178 **High** 179

0272-1732/90/0600-0005$01.00 © 1990 IEEE

Micro News

Send information for inclusion in Micro News one month before cover date to Managing Editor, IEEE Micro, *PO Box 3014, Los Alamitos, CA 90720-1264.*

Technology updates

No time to track new technology? Let *Micro* do it for you.

Chips

AT&T recently developed a deep UV-resist chemical that helps produce circuits as thin as 0.3 µm. Bell Labs invented a version of the chemical that is used with ultraviolet light. Sematech uses the resist to create chips; Olin Corp. will make and sell the chemical.

Hyundai Electronics, Metaflow Technologies, and LSI Logic expect their Lightning Sparc chip to exceed 80 MIPS on most compiled programs. No exotic process technology accounts for this claim. Rather, the device will execute simultaneous, multiple instructions out of order and on speculation beyond unresolved branches. The design maintains strict compatibility with the Sparc instruction set and software.

Transistors

IBM scientists announced a silicon PNP bipolar transistor that switches at 75 billion times per second. The heterojunction bipolar transistor uses an alloy of silicon and a controlled, layer-by-layer percentage of germanium to enhance electrical properties of the critical region.

Electrons

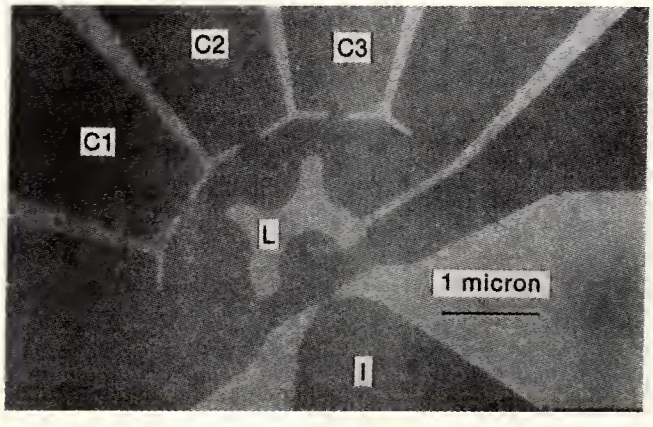
IBM researchers also demonstrated that fast-moving ballistic electrons can be focused and steered as they travel at very low temperatures through gallium arsenide. These electrons travel at about 1 million mph—an interesting speed for circuits. They move ballistically through an impurity-free region called two-dimensional electron gas.

Scientists applied a differential voltage across the gates to steer the electrons about 60 degrees off the original path for as long as 2 µm. (See figure.)

Superconductivity

Researchers at the Georgia Institute of Technology greatly increased deposition rates in the production of superconducting thin films. W. Jack Lackey and his team use a new form of the chemical vapor deposition process to coat flexible fibers with a thin film of superconducting material. They produce 50 to 200 µm of coating per hour during a deposition run, rather than the usual 1 or 2 µm.

What's the angle? The team sends finely ground powders into the reactor rather than evaporating materials and channeling them through the furnace with gas.



IBM researchers introduced electrons through an injector (I), focused the electron beam with a small metal lens (L), and collected the electrons in three areas (C1, C2, and C3).

Silicon Glen: The European Challenge

Christine Miller, Staff Editor

Most people know about the concentrations of electronics firms on Massachusetts Highway 128 and in California's Silicon Valley. Perhaps we know less about the growing international colony of about 300 electronics firms in Scotland called Silicon Glen. What with *glasnost*, *perestroika*, and the opening of European Economic Community markets, *IEEE Micro* wanted to investigate what was happening in this 70 × 30-mile area that runs through central Scotland.

We asked Robert Crawford, director of the North American Headquarters of the Scottish Development Agency, to help us understand the implications of events in Silicon Glen.

Crawford studied government at Harvard University on a J.F. Kennedy scholarship and received his PhD from Glasgow University in politics. He played an integral role in attracting Sun Microsystems and BBN's manufacturing and research and development divisions to Scotland.

The SDA has provided $4.5 billion worth of investments to Scotland in the last four years, mostly from the US.

One of the signs of a shrinking international globe is that more and more businesses are setting up operations in other countries. One study shows that Scotland is the favorite site for US electronics companies to develop European operations. What is attracting them?

High-tech corporation executives perceive Scotland to offer manufacturing excellence and high productivity levels. In addition to major financial incentives such as lower taxation, the country offers a supportive infrastructure for the manufacturing and design of silicon chips. Companies can source raw silicon and a number of chemical supplies and other materials necessary to the semiconductor industry, including masks.

North American personnel are also attracted by the similarities of language and culture, although there are some trade-offs.

What US electronics firms have set up business in Silicon Glen?

IBM arrived first. It's been here so long that people look at the building and think of it as a local firm. Digital followed not long after in the 1960s. The 70s and 80s brought a new wave of semiconductor companies, and the pace accelerated this last year. Probably the main manufacturers of chips are National Semiconductor, Motorola, Seagate, Hughes, and Digital.

These companies came to serve the European market and now realize that 1992 will have serious implications for standards, content, and distribution of their products.

How are they positioning themselves for the European market of 1992?

By being there. They can gain representation in Brussels and directly influence their industry by lobbying the European standards-setting authority in person. The EEC is establishing regulations having to do with local production and content.

The most recent regulations define a product's origin as the place in which the "final substantial" manufacturing phase occurs. In addition, products must have a 40-percent local content level to be labeled "EEC-origin" to avoid the tariffs into EEC countries, which are estimated at 8 or 9 percent.

How do you define this final substantial manufacturing stage?

The most controversial EEC rule of origin concerns the manufacturing of semiconductors. Manufacturing a microchip involves three steps: creating the silicon wafer, printing the semiconductor circuits on the wafer (or diffusion), and cutting the wafer into individual chips. According to the EC (European Commission) 1985 regulations, the country in which the cutting process occurs should be considered the place of origin.

However, as a result of antidumping proceedings against Japanese chip manufacturers, the EC passed 1989 legislation that declared the cutting phase to be only an assembly process. To be exempt from antidumping duties, the diffusion phase of a microchip must now occur in an EEC member country. Any company in the US that looks at trading with Europe should very carefully consider setting up wafer fabrication plants *in* Europe.

Are we talking strictly about manufacturing, or design and development of chips as well?

Both. Local universities offer a strong design base. Most of them have large centers dedicated to semiconductor design.

How large an electronics market exists for the EEC at present?

We project that the European Common Market for electronic components such as ICs will be $37 billion in 1990 and $43.1 billion in 1993. Europe—with a population of 320 million people—buys a lot more from the US than the US does from Europe. We anticipate a drop in the trade balance.

How does the recent acceptance of East Germany into the EEC affect these figures?

At the moment, it's premature to say. It's a medium- to long-term opportunity. People are rubbing their hands with glee over the anticipation of a much larger market.

What kind of applications are Silicon Glen companies currently serving?

The applications are global, although the field of telecommunications is growing tremendously. This is due to the standardization of telecommunications in Europe. Electronics companies in Scotland also have a large customer base in Great Britain, which is the fastest growing market in Europe for PCs. Ranking third in applications is automotive electronics.

In general, the electronics marketplace growth is greater than in the US. Companies can grow their market share.

How are things going recently in this market?

It is healthier than in the US. Actually, we've experienced an absolute avalanche of activity—a flurry of investments—by American companies in Europe. Motorola just invested $160 million in a plant that will produce cellular telephones. NEC is investing more money in its DRAM plant for a total of $246 million. Hughes makes ASICs here. It's the same as in the US—the only difference is that they're making products for Europe.

I suppose Silicon Glen companies spend some time in Brussels.

You bet they do. Standards for electronics are being introduced only by certain committees in Brussels. You have to have company presence in an EEC country to participate in a committee. There are thousands of standards—or "essential requirements"—being established so that products with an EEC seal of approval can be marketed freely among all members of the community.

Establishing all this will take some time. It's not just something that will happen in 1992. The companies that are in Europe, whether they are US, Japanese, or European, whatever, are the ones that influence these committees on the adoption of standards. Their opinions are the only ones that the committee formally notes.

Is European presence the only way to participate?

Another way to have some effect is to lobby through a US trade association. The US Department of Commerce has published an interesting booklet that explains a number of things about standards. It's called *EC 1992: The Commerce Department Analysis of the European Community Directives*, edited by Deborah Miller, and available through the US Dept of Commerce, International Trade Administration, Washington, DC.

What's new with Posix?

Uniforum, the International Association of Unix System Users, has published a white-paper snapshot of the IEEE Project P1003.2 (Posix.2). *Posix Update: Shell and Utilities*, authored by working-group chair Hal Jesperson, targets programmers and managers of technical departments who will be conforming to the new standard. The Uniforum technical committee plans to update and expand this document when Posix becomes a standard.

Bulk copies of 15 or more cost $1 each plus shipping and handling. Contact the Uniforum association at (408) 986-8840. *Your Guide to Posix* and *Posix Explored: System Interface* are also available at that number.

Battle notes from the CISC/RISC wars

Workstation Labs Khornerstone ratings placed several CISCs above RISC competitors in a 21-test benchmark suite. The ALR Powercache 33/4 (based on the 33-MHz i486) turned in a particularly strong performance. The AST Premium 486/25 (based on the 25-MHz i486) and the HP 9000/345 (based on Motorola's 68030) also outdid selected RISC machines.

Phil Magney, director of sales and marketing for Workstation Labs, feels that high-end PCs are as strong as RISC workstations in terms of raw performance. He points out that the floating-point performance critical to scientific applications compares well between the two machines.

He adds, "We have witnessed an incredible comeback in the performance of CISC architectures. However, this situation is temporary. RISC will regain its lead, particularly with the new IBM architectures like the System/6000. We expect a significant increase in performance."

continued on p. 79

Multiplexed Buses: The Endian Wars Continue

Big-endian and little-endian processors can coexist on the same bus, but they need new compiler data types to simplify how they interchange data. New peace terms in the ongoing endian wars propose these data types.

Cohen reported the first skirmishes of the big-endian versus little-endian wars in "On Holy Wars and a Plea for Peace."[1] He described the battles between the big- and little-endians in Swift's *Gulliver's Travels*[2] and related these war stories to the architectural battles between the designers and users of both microprocessor architectures.

Early in the development of the IEEE Futurebus standard,[3] Borrill had documented the difference between what he termed a *type-1 bus* and a *type-2 bus*.[4] These buses multiplex the first data byte (which has the lowest address) with the least and most significant portions of the address, respectively. Borrill observed that the type-2 ordering simplified the interpretation of memory dumps. He therefore proposed that future multiplexed bus standards (with multiple-width data transfers) use type-2 storage formats.

Shortly after Borrill's observations, Cohen—in a beautiful analogy based on *Gulliver's Travels*—defined the differences between big-endian and little-endian processors. Big-endian microprocessors place the first byte of a sequential data stream into the more significant part of a register, while little-endian microprocessors put the first byte into the least significant part of a register. Cohen did not, however, differentiate between the order of bytes within a register and the order used to multiplex data bytes on the bus.

Since Cohen's original plea for peace, Kirrmann has proposed some other terms.[5,6] He observed that the bus-interface hardware could perform byte swaps to automatically convert a big-endian memory access into a little-endian memory access (or vice versa). Since byte swapping was easier to perform on big-endian processors, Kirrmann proposed that external data should have a little-endian byte order.

The byte-swapping algorithms proposed by Kirrmann are dependent on the size of the data that is accessed. The dynamic byte-swapping algorithms fail when hardware misinterprets an access of an unaligned "quadlet" (4 bytes) as two quadlet (2-byte) accesses. (An unaligned quadlet is a 4-byte datum whose address is not a multiple of its 4-byte size.) These algorithms also fail when an on-chip cache isolates the bus interface hardware from the microprocessor's knowledge of the data size.

David V. James
Apple Computer

0272-1732/90/0600-0009$01.00 © 1990 IEEE

The hardware byte-swapping proposed by Kirrmann would have converted big-endian load/store instructions into little-endian load/store instructions. All applications, compilers, and operating systems would have to change accordingly. Big-endian system vendors (who felt their original byte order was correct) were reluctant to provide these extensive software changes.

As you may recall, the bloody civil war in Lilliput resulted from a law requiring inhabitants to break their eggs only at the little end. Eleven thousand Lilliputians died when those who broke their eggs at the big end fought the proclamation. Our big-endians from Lilliput took refuge on the island of Blefuscu. They interpreted Kirrmann's peace proposal as a demand for unconditional surrender and therefore rejected it. Since then, attaching big- and little-endian microprocessors to separate buses has minimized endian war casualties. For example, big-endians use the VMEbus (ANSI/IEEE Standard 1014[7]), and little-endians attach to the IBM PC AT bus or Multibus II (ANSI/IEEE Standard 1296[8]).

However, both big- and little-endians plan to use the active Futurebus+ (IEEE P896.1[9]), SCI (Scalable Coherent Interface, IEEE P1596), and Serialbus (IEEE P1394) draft/proposed standards. The fact that multiple bytes of address and data are multiplexed on Futurebus+ has revived the debate between type-1 and type-2 data storage, as originally defined by Borrill. The new spirit of *glasnost* among world leaders should stimulate engineers to review their own endian ordering issue.

We could define two ordering options for each new bus standard in which each option is optimized for big- and little-endian processors, respectively. Or we could define only one ordering option to simplify the interconnection. Although this solution would be suboptimal for some processors, it would also dramatically reduce the number of product options as well as user confusion.

To resolve the bus-multiplexing issue, we must recognize that the *byte order within integers* and the *byte order during transmission* can differ. Therefore, while the big and little adjectives describe the order of bytes within integers, we use the acronyms Mad and Sad to describe the order of bytes during transmission on a multiplexed address/data bus. To understand this distinction, and with unfailing respect for Swift, we will revisit the communities of Lilliput and Blefuscu, where eggs now ship in Mad- and Sad-endian containers.

After reviewing the endian ordering issues, we conclude that both big- and little-endians can use the same bus standard. For high-performance serialized buses, the Mad-endian order seems superior to a Sad-endian order. For consistency between serialized and multiplexed parallel buses of various widths, I propose the Mad-endian order for future multiplexed standards. To minimize the interface costs to Mad-endian buses, I also propose a big-endian order for shared data.

In the spirit of *glasnost*, we should recognize that big- and little-endian processors will continue to coexist and therefore should easily exchange data with the opposite endian order. For convenience, we should extend the common operating-system languages (C, for example) to accurately specify the order (big or little) and the size (1, 2, 4, 8 bytes) of shared data types. With such language support, the compiler (rather than the programmer) can perform the proper byte-swapping operations. In the absence of such changes, the new extension of C called C++ allows the user to easily (but less efficiently) define such data classes.

As background for this article, I start by reviewing a Lilliput travel report and the origin of big- and little-endians and extend Lilliputian history to define Mad- and Sad-endians as well.

Endian terminology

After being drafted into recent (largely) US endian wars, I have struggled for clear answers to this seemingly complex and highly religious problem. In frustration, I accompanied a special task force that visited the communities of Lilliput and Blefuscu to learn the history of their successfully negotiated truce.

Lilliput trip summary. In large part, the truce in Lilliput can be attributed to the new leader Vehcabrog Retsim. He negotiated a truce based on the principles of *glasnost* and *perestroika*. Although his motives were largely altruistic, he was also concerned with defections of Lilliputians to Blefuscu, where the prices of eggs and fruit were lower. In our discussions with Vehcabrog, we realized that the truce was heavily influenced by two industries. The local egg industry had introduced styrofoam egg cartons, while the import fruit industry had adopted light-weight fruit containers.

Egg containers. The introduction of styrofoam egg cartons forced the Lilliput congress to struggle with a difficult decision. Should the eggs be inserted with the little or the big end at the top? The members debated the issue for weeks! The religious zealots insisted on having eggs packed small end up so that the egg-extraction and cracking orders were maintained (the little end is always first). The shipping lobby preferred to pack eggs with more of the egg up to minimize shipping losses.

With their Christmas vacations approaching, the congress legislated a two-carton system. The Mad-endian carton puts *m*ost of the egg on top, next to the *ad*dress label. The Sad-endian carton puts less of the egg (the *s*mall part) on top, next to the *ad*dress label. This system maintained consistent order, since *m* appears before *s* in the alphabet and *b* appears before *l*.

Both cartons labeled the egg positions and included an international logo so the type of carton could be

recognized from a distance. The system classified individuals by how they pack their eggs (Mad- or Sad-endians) and the way they crack their eggs (big- or little-endians). (See Figure 1.) This caused a boom for the T-shirt industry, which created the four logos shown at the beginning of this article.

Fruit imports. The export market to Lilliput and Blefuscu improved because these packages were standardized. Certain California fruit exporters faced a serious dilemma. They had an existing shipping container based on a little-endian design but shipped most of their fruit to big-endians on Blefuscu. Although some round fruit could be packed in either order, pears and Delicious apples bruised when packed in the opposite order. Concerned with the time needed for redesigning another package, the California exporter modified the address label instead. Several clever designers observed that fruit could be extracted in big-endian order if the bottom of the box was opened first. The same package could now be used for shipments to Lilliput and Blefuscu, but a special SBE (Sad big-endian) label appeared on shipments to the big-endian refugees on Blefuscu[10] as shown in Figure 2. The refugees learned to flip the package (label down) so that the fruit was in big-endian order when they opened the top of the package to take a bite of fruit (Figure 2b). Although the SBE label was initially confusing, the independent grocers on the island of Blefuscu soon accepted the packages. The California export business boomed.

The Lilliputian solution. After several months, shippers minimized the amount of cracked eggs and bruised fruit by shipping the Sad-endian packages with the big end up. However, this placed the address on the bottom of the Sad-endian packages, which increased shipping delays. The shippers eventually influenced their suppliers and, in the spirit of *perestroika*, the little-endians on Lilliput agreed to use new packages based on the superior Mad-endian package design. Note that eggs—still cracked at the same end—are immediately flipped by the little-endians when removed from the Mad-endian package. Also, the Blefuscu grocers still flip their shipments of apples in SBE packages before customers extract their first bite.

The processor analogy. Extending Cohen's analogy, I now use the words big, little, Mad, and Sad to describe computing systems. The terms *big* and *little* describe the mapping between data-byte addresses and byte locations within a register. The terms *Mad* and *Sad* describe the mapping between data-byte addresses and byte-lane positions on the bus (see box on the next page). To clarify the distinction between big- and little-endian ordering (as defined by the processor instruction set) and Mad- and Sad-endian ordering (as defined by the interconnection standard), let's observe the movement of data during a transfer from the disk into a processor register. The following steps are involved:

1) Disk to memory. The system copies data from disk through disk controller to memory (for clarity, assume a DMA transfer).

2) Memory to cache. On a cache miss, the system copies a 64-byte sector from memory to cache.

3) Cache to processor. After the cache sector is valid, the system copies data from the cache to the processor's register in the CPU.

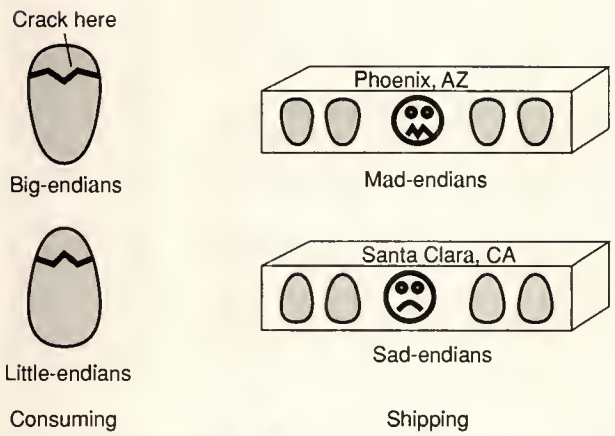


Figure 1. Endian types.

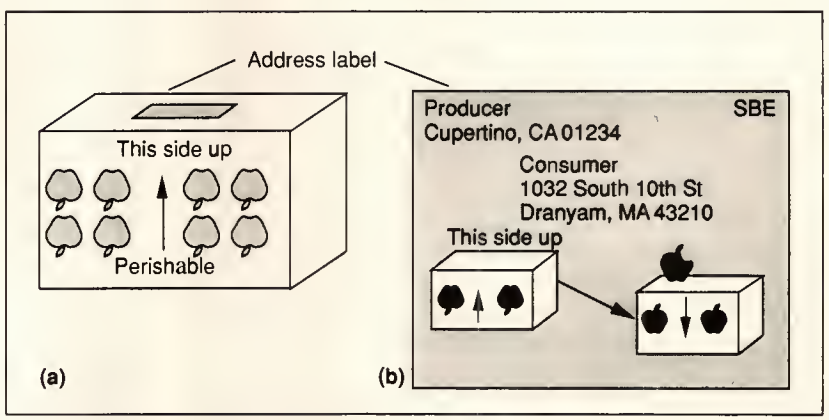


Figure 2. California's export package (a); enlarged shipping label (b).

Glossary of Terms

To distinguish between the processor and bus ordering conventions, I define the common data objects and the processor types.

Byte (or octet): 8 bits of data
Doublet: 2 bytes of data
Quadlet: 4 bytes of data
Octlet: 8 bytes of data, different from octet (8 bits of data)
Big-endian processor: When data is loaded into a multibyte register, the first byte (with the lowest address) is the most significant byte of the data.
Little-endian processor: When data is loaded into a multibyte register, the first byte (with the lowest address) is the least significant byte of the data.

Separate terms describe the relative order of address and data when they are multiplexed on the bus (in space or time).

Mad-endian bus: When data transfers on a multiplexed address/data bus, the first byte (with the lowest address) is multiplexed with the more significant byte of the address.
Sad-endian bus: When data transfers on a multiplexed address/data bus, the first byte (with the lowest address) is multiplexed with the least significant (small) byte of the address.

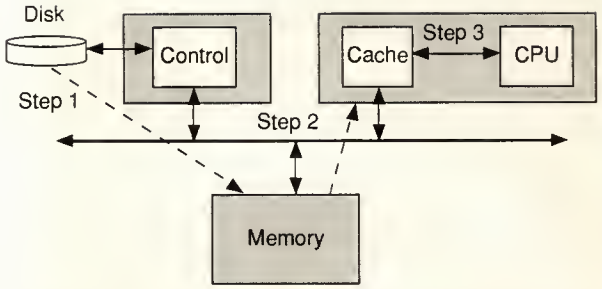


Figure 3. Disk-to-register transfers.

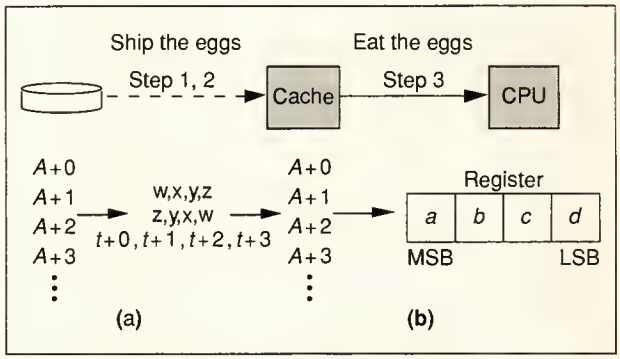


Figure 4. Disk-to-register orderings for Mad- and Sad-endian bus standards (a) and big- and little-endian processors (b).

Figure 3 illustrates the steps. These transfers are logically equivalent to another two steps:

1) Disk to cache. The system copies the data sector from disk to cache, using the byte lanes or time slots defined by the bus standard.
2) Cache to CPU. The system copies some of the cached data to a processor register, using the address-to-register map defined by the processor's instruction set.

Figure 4 illustrates these logical transfers (numbers 1, 2, and 3 refer to the transfers in Figure 3). The bus specification (or specifications, if crossing a bridge) defines the mapping between increasing data-byte addresses ($A+0, A+1, \ldots$) and data-byte lanes on the bus (in space or time). A parallel bus (like the 32-bit-wide version of Futurebus+) specification defines a mapping between increasing byte addresses and the spatial byte lanes (w, x, y, z). In Serialbus and the serialized version of SCI, sequential byte addresses are mapped to sequential time slots: $t+0, t+1, t+2$, and $t+3$. Of course, all interfaces on the same bus use the same convention to avoid scrambling the data.

Although connecting big- and little-endian processors to Mad- and Sad-endian buses, respectively, can minimize interface costs, either type of processor can connect to either type of bus. The CPU's instruction set defines the mapping between the addresses in the cache and the byte positions within a register (a, b, c, d). The byte order used for data transmission (Mad- or Sad-endian) need not be the same as the byte order used within the processor (big- or little-endian). Either type of processor can connect to either type of bus, but the interface costs are less when big- and little-endian processors connect to Mad- and Sad-endian buses.

Big and little CPUs

Now that we have the travel report from modern Lilliput and a new set of endian names, let's review the history of little- and big-endian computer architectures. We find humor here as well.

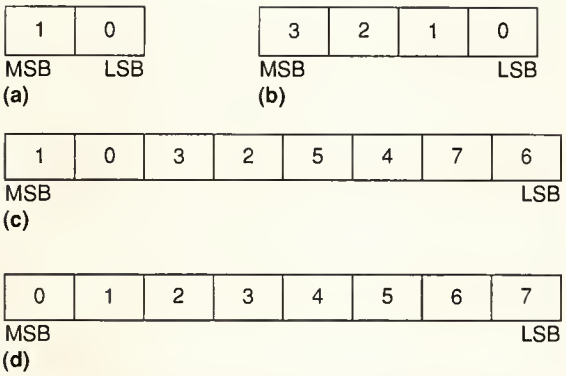


Figure 5. VAX data formats: doublet integer (a), quadlet integer (b), octlet floating point (c), and numerical strings and packed decimal (d).

Inconsistent byte orderings. As Cohen illustrated, many little-endians (including myself) originally ate bytes and nibbles preprocessed by the VAX. We therefore called ourselves little-endians. When standard presentation formats are used (most significant items on the left), the inconsistency of this religion becomes clear, as shown in Figure 5. In this figure, the numbers represent the address offsets of data bytes, measured from the base address of the data structure.

Graphic artists can and do hide these inconsistencies in the second dimension.[11,12] Figure 6 is an example of these creative VAX illustrations. Fortunately, compilers hide most of these inconsistencies from application programmers. For illustrative purposes, Figure 6 shows how the BCD (binary-coded decimal) string $914.17 and the ASCII character string CAT are stored. Some little-endians call this architecture fully consistent, since 2-byte and 4-byte integers have the same little-endian order. This claim reflects design improvements since the introduction of the PDP 11, which uses a mixed endian order for 4-byte integers.[13]

Inconsistent bit and byte orderings. Big-endians commit different sins. As children, their first bits and nibbles came from ALU chips, which map zero (0) to the least significant bit. From the diet of their youth, some (otherwise) big-endian designers insist on using the little-endian notation to describe bits and the big-endian notation to describe bytes. Figure 7 illustrates the inconsistency of this notation.

Confusion abounds when the two data types meet (for example, at the boundary of a bit-extract instruction). I suspect that this inconsistency caused the different sets of bit-manipulation instructions on the 68000 family processors from Motorola.[14] In the original single-bit-manipulation instructions (which are implemented on all 68000s), the zero is placed on the least significant bit (LSB). In the more recent multiple-bit-

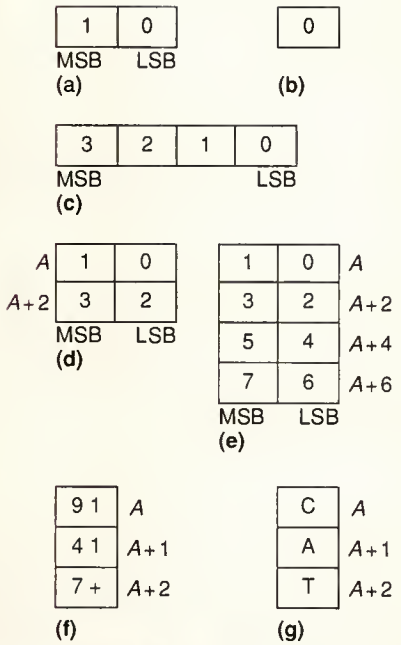


Figure 6. Creative VAX illustrations of formats: doublet integer (a), byte integer (b), quadlet integer (c), quadlet floating point (d), octlet floating point (e), packed decimal (f), and character string (g). (Reproduced with permission from Digital Press/Digital Equipment Corporation.[11])

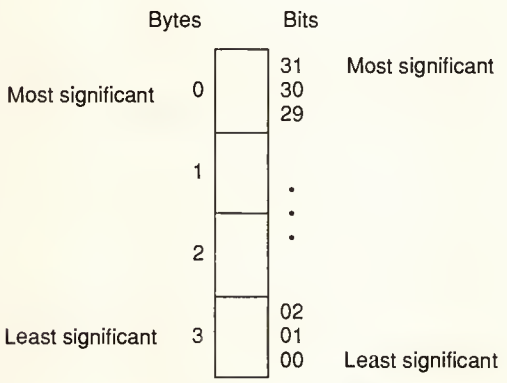


Figure 7. Inconsistent 68000 family labels.

manipulation instructions (on the 68020, 030, and 040), the zero label is put on the most significant bit (MSB). The consistency of the more recent labeling convention was needed to describe bit fields that cross byte or word boundaries.

Note that IBM (on the S/360 and 370[15]) and Hewlett-Packard (on the PA-RISC processor[16]) consistently map zero to the MSB.

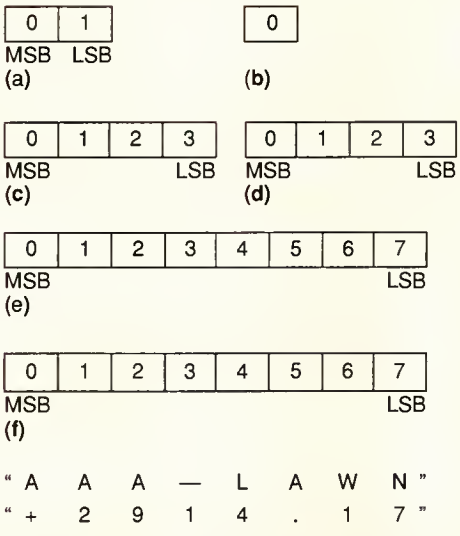


Figure 8. Consistent big-endian formats: doublet integer (a), byte integer (b), quadlet integer (c); quadlet floating point (d); octlet floating point (e), and numerical strings and packed decimal (f).

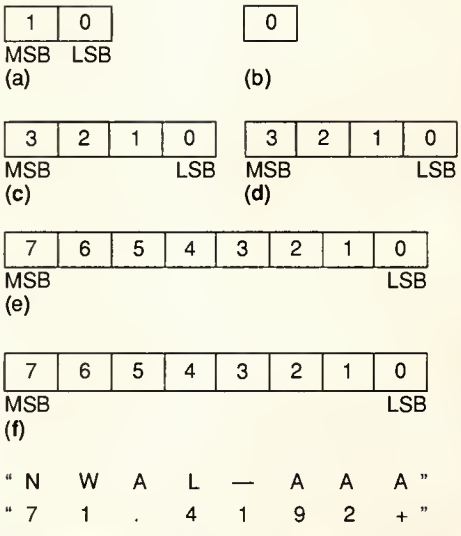


Figure 10. Largely little-endian formats: doublet integer (a), byte integer (b), quadlet integer (c), quadlet floating point (d), octlet floating point (e), numerical strings and packed decimal (f).

```
/* shift data from "address" to shifted-byte location */
#define get(address,left_byte_shift)\
   (*(address)<<8*(left_byte_shift))
#define byte char

/* Extract from a big-endian instruction simulator */
execute_big(instruction)
{ unsigned byte *a;
   /* calculate data address */
   a= address_calculate(instruction);
   /* select destination register */
   reg=destination_register(instruction);
   /* switch and execute op-code */
   switch(instruction.op_code) {
   case LOAD_UNSIGNED_BYTE:
      GR[reg]= get(a+0,0);
      break;
   case LOAD_UNSIGNED_DOUBLET:
      GR[reg]= get(a+0,1) | get(a+1,0);
      break;
   case LOAD_UNSIGNED_QUADLET:
      GR[reg]=
        get(a+0,3) | get(a+1,2) | get(a+2,1) | get(a+3,0);
      break;
   .
   .
}
```

Figure 9. Big-endian simulation code.

Consistent big-endian architectures. After zero is mapped to the MSB of a byte and to the most significant byte of a word (for some, an initially unnatural act), the consistent big-endian format is easily derived. Figure 8 presents these formats. The values on the bottom of the figure ("AAA-LAWN" and "+2914.17") illustrate how ASCII and BCD strings are stored.

Most big-endian processors specify this consistent byte order, with the exception of the BCD sign digit. These processors include the MIPS R2000[17] (in big-endian mode), the Sun Microsystems Sparc architecture,[18] the IBM S/360 and 370,[15] the PA-RISC processor, and the Motorola 68000 family[14] and 88000.[19]

On the otherwise big-endian processors (S/360 and 370 and PA-RISC), the BCD sign digit (which is most significant) comes last. This is an historical exception, which originally simplified processing of variable-length BCD strings (that have since migrated to fixed-length formats).

The register layouts are useful for hardware designers, but instruction-set simulators best describe an architecture to system programmers and compiler writers. From their software perspective, a big-endian instruction places the first byte (with the lowest address) in the more significant part of a register, as illustrated by the code segments in Figure 9.

Largely little-endian architectures. European and US designers speak and write in big-endian dialects—the first character or digit is the most significant. Thus, all little-endian architectures have at least one inconsistency. For the remaining integers and floating-point numbers, designers can develop a consistent little-endian architecture, as shown in Figure 10.

This byte order is specified by the most consistent little-endian processors, including the R2000[17] (in little-endian mode) and the Intel i860[20] and i960.[21]

The apparently reversed order of character strings generates little-endian jokes (including several in this article). The inconsistencies are also visible in memory-image dumps. When printed in the same order, either the character strings or the integers appear reversed. Good little-endians generate programs that dump data in either order according to user preference.

From a software perspective, a little-endian places the first byte (with the lowest address) in the least-significant part of an integer or floating-point register, as illustrated by the code segments in Figure 11.

A big/little comparison. Now that we have an accurate definition of big- and little-endian processors, let's review the software advantages of big and little processor architectures. Historical reasons for choosing a big or little architecture include:

1) Byte-wise integer additions (little). With byte-sequential additions, a little-endian processor can start the addition while data is being fetched in order (*A*, *A* + 1, etc.).

2) Fast BCD arithmetic (big). For BCD arithmetic, a big-endian processor is faster; the system can compare multiple digits by using one integer-compare instruction.

However, in VLSI designs, the transmission order from memory no longer affects the speed. Also, few RISC processors directly support BCD arithmetic, since BCD data can be quickly converted into binary formats, processed in binary form, and converted back into BCD formats.

Functional criteria for choosing a big-endian or little-endian processor architecture include:

1) Character-string sorting (big). When it compares integer-aligned character strings, a big-endian processor is faster (the integer ALU can compare multiple bytes in parallel).

2) Decimal/ASCII dumps (big). On a big-endian processor, all values can be printed left to right without causing confusion.

3) Integer address conversion (little). When it converts a quadlet integer address to a doublet integer address (to use the LSBs), a big-endian processor has to perform addition.

4) Fraction address conversion (big). When it converts a quadlet fraction address to a doublet fraction address (to use the MSBs), a little-endian processor has to perform addition.

5) Floating-point compares (consistent). If endian order is consistent, an integer ALU can compare floating-point numbers.

However, simple character-string sorting becomes less important as library searches and foreign-language sorts use more complex compare routines.

```
/* shift data from "address" to shifted-byte location */
#define get(address,left_byte_shift) \
  (*(address)<<8*(left_byte_shift))
#define byte char

/* Extract little-endian instruction simulator */
execute_little(instruction)
{ unsigned byte *a;
  a= address_calculate(instruction);
  reg=destination_register(instruction);
  switch(instruction.op_code) {

  case LOAD_UNSIGNED_BYTE:
    GR[reg]= get(a+0,0);
    break;
  case LOAD_UNSIGNED_DOUBLET:
    GR[reg]= get(a+0,0) | get(a+1,1);
    break;
  case LOAD_UNSIGNED_QUADLET:
    GR[reg]=
      get(a+0,0) | get(a+1,1) | get(a+2,2) | get(a+3,3);
    break;
  .
  .
}
```

Figure 11. Little-endian simulation code.

Notational convenience favors big or little architectures for the following reasons:

1) Consistent order (big). The big-endian processors store their integers and character strings in the same order (the most significant *byte* comes first).

2) Integer notation (little). For little-endian processors, the bit-index values closely match the following integer definition:

$$\cdot\, b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

3) Fraction notation (big). For big-endian processors, the bit-index values closely match the signed fraction definition:

$$- b_0 \times 2^0 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} \ldots$$

The little-endian integer notation was convenient for early microcomputers, whose (integer-like) addresses often outgrew the size of their containers. However, because micros are maturing to 64-bit addresses, this is less of an issue.

To an unbiased compiler writer, the best software architecture is not clear, since the weighting assigned to each of these criteria differs from engineer to engineer. The consistency of the big-endian ordering makes it the (slightly) preferred software solution, but the bus-interface costs (which are discussed in the following sections) influence this judgment as well.

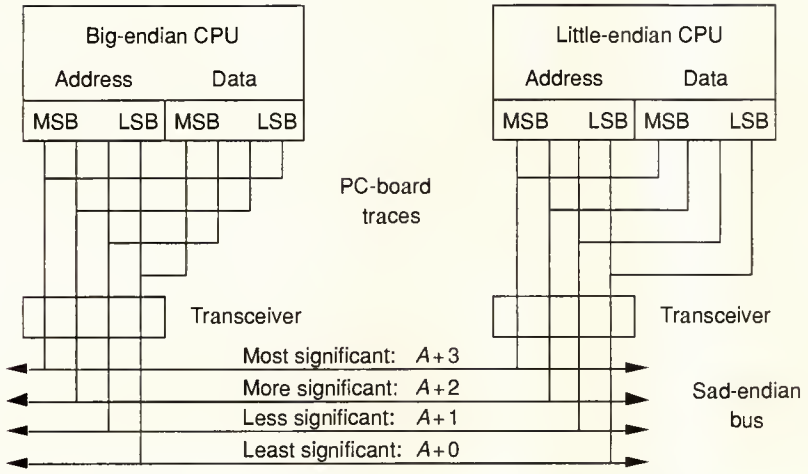


Figure 12. Demultiplexed CPUs on Sad buses.

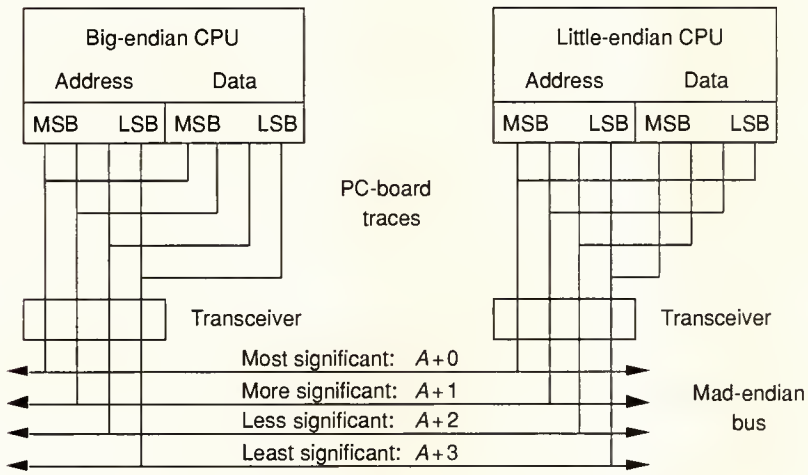


Figure 13. Demultiplexed CPUs on Mad buses.

Sad and Mad buses

Several RISCs have a byte-order switch (big-endian or little-endian) that is set during system initialization[17] by a control register[20] or by bits in the page table.[21] In demultiplexed processor designs, the single big/little bit supports any combination of (Mad/Sad and big/little) system designs,, since the PC-board traces and bus transceivers specify Mad- or Sad-endian ordering.

Demultiplexed CPUs. Figure 12 shows the mapping of address and data for the Sad-endian Nubus (IEEE Standard 1196[22]) and Multibus II. The groups of eight address/data signal wires carry the labels of their address significance (most through last) and the address of data bytes ($A + 0$ through $A + 3$). Although mostly little-endians have attached to the Sad-endian Multibus II standard, mostly big-endian processors have attached to the Sad-endian Nubus standard. For example, Apple's Nubus-based Macintosh II products use a big-endian 68020 processor.

Figure 13 illustrates a similar mapping of address and data to a Mad bus standard. Unfortunately, the endian order of the processor is hardwired by the PC-board traces and cannot be dynamically changed in the field. Additional byte-swapping logic would support dynamic changes between big- and little-endian processing options.

Multiplexed CPUs. As dynamically configurable CPU chips migrate to multiplexed address/data buses, two bits are necessary to specify the four design environments (Mad/Sad and big/little). Figure 14 illustrates these four mappings for a 2-byte store to a 4-byte-aligned address on a 32-bit multiplexed address/data bus where the MSB and LSB labels apply to the address bits. Two unique hexadecimal digits (0 and D or 1 and F) in the relevant byte locations unambiguously specify the correct byte-lane mappings. The processor registers and bus data path are assumably both 32 bits in size; 64-bit extensions are an exercise for the reader.

Data sharing

Having shown how big- and little-endian processors can connect to the same bus, let's discuss how their data is shared in a distributed multiprocessor environment.

Conversion macros. Consider processing time-sequential data generated by processor *X* and filtered by processor *Y*. The data has a standard header that specifies the data type (big/little, integer/floating point, and size) and the number of data elements.

With the data type properly identified, processor *Y* can correctly process data generated by processor *X* in either order. Of course, data format conversions are needed when the internal and external orders differ, as illustrated by the code segments in Figure 15.

Note that data processing depends on the data order

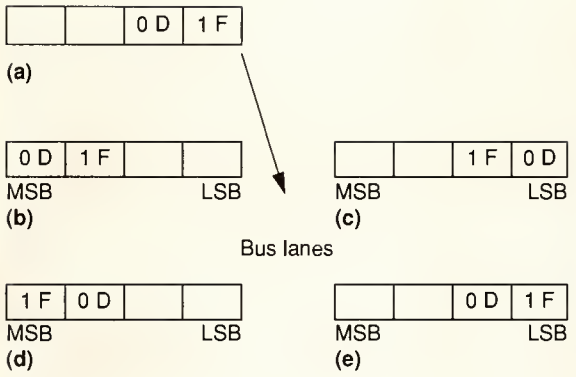


Figure 14. Multiplexed CPUs on Mad and Sad buses: CPU register (a), Mad big-endian (b), Sad big-endian (c), Mad little-endian (d), and Sad little-endian (e).

(big or little), its size (byte, doublet, or quadlet), and the meaning of the data (unsigned integer, signed integer, or floating point). These three parameters are necessary to accurately specify external data types.

The coding fragments in Figure 15 are similar to those contained in audio and video signal-processing routines.[23] The Berkeley TCP-IP (transmission control protocol-internet protocol) Unix source code on the VAX provides similar byte-swapping conversion routines (htonl, htons, ntohl, and ntohs).

New data types. Most languages, including C, assume that data is processed within the domain of one processor. However, the boundaries between processor domains become blurred when systems are connected through bus bridges or when several processors share the same bus. Therefore C (and other byte-level programming languages) should define portable data types as well as the currently defined host-specific data types.

Table 1 on the next page specifies several new C data types. Since each language has a different set of name and syntax constraints, the actual language specification may differ.

If the type or order is not specified, the compiler assumes the most efficient values. Note that the integer size specification should extend to support 64-bit integers, which are necessary to hold the 64-bit addresses (currently being defined by Futurebus+ and SCI) or binary versions of large BCD values.

Standard big- and little-endian floating-point data bytes need definition as well. The IEEE 754 standard[24] defines the significance of the bits in the floating-point number. I propose that the standard define consistent big- and little-endian mappings that map the lowest byte address to the most and least significant components of the floating-point value.

```
/* these define statements work on many machines */
#define byte char                /* 1-byte integer */
#define doublet short            /* 2-byte integer */
#define quadlet long             /* 4-byte integer */

#define BIG 1                    /* if a big-endian */

/* move byte between "from" and "to" byte-lanes, */
#define mov(data,from,to) (((from-to)>=0 ?\
   (data)>>8*(from-to):(data)<<8*(to-from))&(0XFF<<8*to))

#define swap2(data) (mov(data,1,0) | mov(data,0,1))
#define swap4(data) (mov(data,3,0) | mov(data,2,1) \
   mov(data,1,2) | mov(data,0,3) )

/* native_to_little, ... , big_to_native conversions */
#define ntolu2(data) (BIG ? swap2(data) : (data))
#define ntobu4(data) (BIG ? (data) : swap4(data))
#define ltonu2(data) (BIG ? swap2(data) : (data))
#define btonu4(data) (BIG ? (data) : swap4(data))

struct dblock {                  /* make 4-byte aligned */
   byte data_type;               /* order and data format */
   byte data_length; /* number entries (256 max) */
   byte other_info[2] /* to keep alignment */
   unsigned byte data[MAX];   /* data follows header */
};


/* Add constant to all elements: "data_value[i]+=add_value;"
 * After recompilation, code runs on many machines */
add_to_any_data(in_ptr,add_value)
struct dblock *in_ptr;           /* data block pointer */
{ /* data element types */
   unsigned integer format= inptr->data_type;
   /* number of elements (127 max) */
   unsigned integer length= inptr->data_length;
   /* data element pointer */
   unsigned byte *bptr= inptr->data;
   unsigned integer count;

   /* step through data elements */
   for (count= 0; count<length; count++) {
      switch(format) {
      case UNSIGNED_BYTE:
         *bptr= *bptr+ add_value;
         bptr+= 1;
         break;
      case LITTLE_UNSIGNED_DOUBLET:
         result= ltonu2(*(unsigned doublet *)bptr) +
            add_value;
         *(doublet *)bptr= ntolu2(result);
         bptr+= 2;
         break;
      case BIG_UNSIGNED_QUADLET:
         result= btonu4(*(unsigned quadlet *)bptr) +
            add_value;
         *(quadlet *)bptr= ntobu4(result);
         bptr+= 4;
         break;
      case BIG_FLOAT_QUADLET:
         ...
      }
   }
}
```

Figure 15. Signal-processing routine (macros specify endian-order conversions).

Table 1.
Integer declaration components.

Order	Type	Size
Big	Signed	Byte
Little	Unsigned	Doublet
Either	Either	Quadlet
—	—	Octlet

```
/* typedefs, for external data */
typedef union {
   unsigned byte int_u1;
   /* unsigned doublet (2-byte) integer */
   little doublet unsigned integer int_ul2,
   /* unsigned quadlet (4-byte) integer */
   big quadlet unsigned integer int_ub4;
} data_types;

/* Add constant to array: "data_value[i]+= add_value;"
 * After recompilation, code runs on many machines */
add_to_any_data(in_ptr,add_value)
struct dblock *in_ptr;          /* data-block pointer */
{  data_types *dptr;
   /* ... setup code, as before */
   for (count= 0; count<length; count++) {
      /* pointer-type conversion */
      dptr= (data_types *)bptr;
      switch(format) {
      case UNSIGNED_BYTE:
         dptr->int_u1= dptr->int_u1+ add_value;
         bptr+= 1;
         break;
      case LITTLE_UNSIGNED_DOUBLET:
         dptr->int_ul2= dptr->int_ul2+ add_value;
         bptr+= 2;
         break;
      case BIG_UNSIGNED_QUADLET:
         dptr->int_ub4= dptr->int_ub4+ add_value;
         bptr+= 4;
         break;
      case BIG_FLOAT_QUADLET:
         ...
      }
   }
}
```

Figure 16. Signal-processing routine (data types specify endian-order conversions).

The new data types simplify the previous signal-processing routine as shown in Figure 16.

In the absence of these supported data types, programmers must define their own conversion routines. The C++ programming-language conversions are user-defined data classes that are convenient and easy to use. However, they would be less efficient than compiler-supported data types.

Byte-swapping performance. To efficiently support the new data types, processor instruction sets should minimize the penalty for swapping between opposite-endian orders. Let's calibrate some designs.

Table 2 lists the number of instructions required to perform a register-to-register byte swap for unsigned, 2-byte and 4-byte values. Since compilers can preserve frequently used constants in registers, the generation of constant register values does not appear in the instruction-count totals.

Note that the rotate and deposit instructions improve the performance of processors over other processors with simple shift instructions. Also, the number of CISC-like instructions on the VAX can be misleading since these instructions may take many clock cycles to execute. Any more detailed conclusions would be premature. Although specific instruction counts are convenient, actual system performance may vary.

Serialized buses

Processors can attach to Mad- or Sad-endian interconnection standards, and data can transmit in big- or little-endian formats. Therefore, which standards should be used to communicate between big- and little-endian processors? Let's explore the serialized versions of bus standards to find the answer.

Transmission order. The developing serialized bus standards SCI and Serialbus propose that data be transmitted in a bit-serial fashion. Since data bytes are physically (or logically) transmitted in a sequential fashion, little controversy occurred in the working groups over the endian-ordering on the bus.

The MSBs of the address always come first in time, so the performance-critical routing decisions (based on the MSBs of the address) can occur before the LSBs of the address have arrived. Data bytes are transmitted to increasing data-byte addresses (for consistency with existing bus and network standards). Figure 17 demonstrates this transmission order.

In the case of SCI, 6 bytes of control information separate the address field into two sections (addr_hi and addr_lo), which influences the packet-routing decisions. The address offset at the target (addr_lo) is delayed, since the less significant part of the address has no effect on the packet's route.

To avoid confusion (and several dialects of bus-interface chips), the bit and byte label notations remain consistent. In all transmissions bit[0] and byte[0] are sent through the interfaces first in time. The initial items are placed on the left of the figure to match the text in the document. Figure 18 shows these notations.

Within a byte, bit zero (b[0]) is the most significant and transmits first. Transfers begin at byte-address zero (measured as an offset from the transfer address) and continue from increasing byte-offset addresses. For

Table 2.
Number of register-register swapping instructions.

Processor	Doublet	Quadlet	Comments
68020,30,40	1	3	Doublet byte swap and rotate
88100	3	5	Extract, deposit, and rotate
PA-RISC	2	3	Deposit and rotate
R2000	4	8	Shifts, And-masks, and Or-merges
Sparc	4	8	Shifts, And-masks, and Or-merges
VAX	1	2	Movtc (Move translated characters) instructions
i486	1	1	Byte-swapping instructions
i860	4	5	Two-instruction double shifts
i960	4	5	Rotates, Ands, and Ors

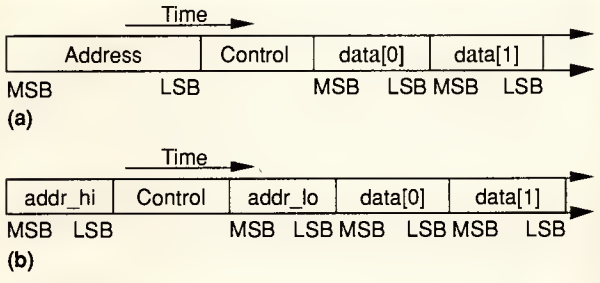


Figure 17. Ordering on Serialbus (a) and SCI (fiber option) (b).

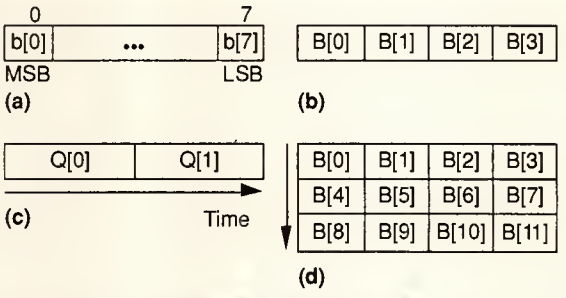


Figure 18. Bit and byte labels in SCI and Serialbus: bits in a byte (a), bytes in a quadlet (b), quadlets in an octlet (c), and bytes in memory (d).

little-endian designers who are used to seeing zero on the right, this notation initially seems unnatural. Thinking of data as ASCII strings rather than numerical integers can partially overcome this reaction.

Note that both of these serialized buses use a clearly superior Mad-endian order. This will continue to be true on other serial buses as well, since the most significant portion of the address is needed first to reduce routing delays.

Byte significance. The bus interface determines the byte significance when data is routed from the interconnection to a multibyte register. For the control and status registers (CSRs) specified by Serialbus and SCI, the bus interace demultiplexes the address and data in the same order. Figure 19 shows this consistent (and therefore lower cost) order.

This consistent byte-demultiplexing order indirectly specifies a big-endian order for the standardized CSRs, which are being defined by the proposed IEEE P1212 CSR architecture standard.

Note that the CSR architecture (which is shared by the proposed Futurebus+, SCI, and Serialbus standards) only standardizes a subset of the registers on a node. Firmware accesses these registers during system initialization. The other registers (which are accessed

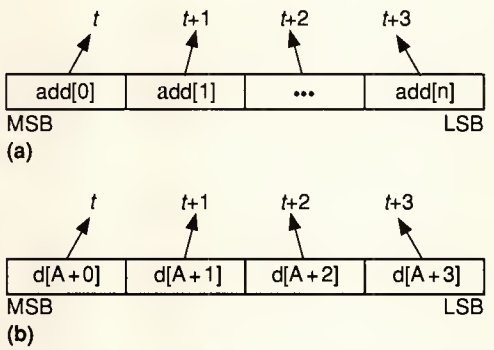


Figure 19. Byte selection timing in SCI and Serialbus for the address register (a) and the CSR data register (b).

primarily by I/O driver software) can have either endian order since their definition is vendor dependent and beyond the scope of the proposed CSR architecture. Thus, the endian order of the frequently accessed registers can still match the endian order of the vendor's instruction set.

What with so many Mad big-endians, Mad little-endians, Sad big-endians, and Sad little-endians, it is easy to become confused. Systems can easily scramble data if the bytes are incorrectly swapped when the data is created, transmitted, or consumed.

It is crucial that interconnecting buses or bridges do not reorder data bytes. In that case, the reordering in the processor becomes more complex and depends on the transport path. This eliminates the use of value-invariant protocols, which (in an attempt to please their users) automatically convert integers between native big- and little-endian formats. The fact that the actual integer width can differ from its assumed value dooms these automatic conversions to failure.

The address-invariant mappings, which are assumed here, would eliminate this problem. The correct interpretation of data by the user remains path-independent. Only the data type assumed by the producer influences the conversion of data at the consumer level.

The industry should stop creating two versions of each new bus standard (as has been done for Futurebus+). Two variants increase development costs and design time and confuse users as well. Buses should pick one (Mad or Sad) solution when they are shared by different (big and little) processors. Although a single ordering option is suboptimal for some designs, this approach reduces the overall system costs.

Serialized bus standards support the Mad-endian order for bus design since it reduces the routing delays through active switches. For consistency between serial and parallel buses, the Mad-endian order should also be incorporated into all multiplexed bus standards that are shared by big- and little-endian processors.

Since the software analysis cannot pick a favored big- or little-endian data order, the order that minimizes the cost of Mad bus designs seems preferable. Therefore, I propose a big-endian data order for standardized data or registers that are shared by big- and little-endian processors.

Languages and compilers should assist system designers, who deal with the relics of the past as well as the dreams of the future. Data types supported by C (and other byte-oriented, systems-programming languages) should have optional big and little data descriptors to override the compiler's default native order.

However, we should respect the privacy of others at all times. Although we need data-interchange standards when data is shared, the actions performed by a processor with its own cache, in the privacy of its own data structures, should remain beyond the scope of our standards.

Acknowledgments

I thank Bill Worley and Michael Mahon (two Hewlett-Packard RISC architects) whose consistent logic initiated my defection from the little-endian to the big-endian camp. I also thank Paul Borrill (IEEE P896 Futurebus+ chair), Bob Greiner (Futurebus+ participant), Dave Gustavson (IEEE P1596 SCI chair), Michael Teener (IEEE P1394 Serialbus chair), Mark Williams (Futurebus+ bridges coordinator), and Randy Weber (Futurebus+ CSR coordinator), who provided valuable inputs. Robert Stewart and reviewers provided the continued encouragement and editorial guidance that transformed a view-graph presentation to the IEEE Computer Society Microprocessor Standards Committee into this article. Many little-endians who prefer to remain anonymous also provided guidance and clarification.

References

1. D. Cohen, "On Holy Wars and a Plea for Peace," *Computer*, Vol. 14, No. 10, Oct. 1981, pp. 48-54.
2. J. Swift, *Gulliver's Travels* (Original title: *Travels into Several Remote Nations of the World*), Putnam Publishing Group, New York, 1948.
3. *IEEE Standard 896.1-1987, Standard for Futurebus Backplane Bus*, IEEE Computer Society Press, Los Alamitos, Calif., 1988.
4. P. Borrill, "Microprocessor Bus Structures and Standards," Vol. 1, No. 1, *IEEE Micro*, Feb. 1981, pp. 84-95.
5. H. Kirrmann, "Data Format and Bus Compatibility in Multiprocessors," *IEEE Micro*, Vol. 3, No. 4, Aug. 1983, pp. 32-47.
6. D. Del Corso, H. Kirrmann, and J.D. Nicoud, *Microcomputer Buses and Links*, Academic Press, New York, 1986, pp. 295-296.
7. *ANSI/IEEE Standard 1014-1987, Versatile Backplane Bus: VMEbus*, IEEE CS Press, 1987.
8. *IEEE Standard 1296-1987, Standard for a Full-Feature 32-Bit Backplane Bus*, IEEE CS Press, 1988.
9. *IEEE Draft Standard P896.1-1989, Standard for Futurebus+*, IEEE CS Press, 1989.
10. Apple Computer, *Macintosh Family Hardware Reference*, Addison-Wesley Publishing, Reading, Mass., Aug. 1988, pp. 22-26.
11. C.G. Bell, J.C. Mudge, and J.E. McNamara, *Computer Engineering, A DEC View of Hardware Systems Design*, Digital Press/Digital Equipment Corp., Bedford, Mass., 1978, pp. 412-413.
12. Digital Equipment Corporation, *VAX Architecture Handbook*, Digital Press, 1981, pp. 31-49.
13. Digital Equipment Corporation, *PDP11/40 Processor Handbook*, Digital Press, 1972, pp. 7-1 to 7-6.
14. *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, Motorola Corp., Schaumburg, Ill., 1987, pp. 2–4 to 2–6.

15. *IBM System 370 Principles of Operation,* Order No. GA22-7000-6, Poughkeepsie, New York, pp. 3-2 to 3-3.

16. *HP Precision Architecture Handbook,* 2nd ed., Hewlett-Packard, Palo Alto, Calif., June 1987, pp. 2-2 to 2-5.

17. G. Kane, *MIPS R2000 RISC Architecture,* Prentice-Hall, Englewood Cliffs, N.J.

18. *The SPARC Architecture Manual,* (Rev. 5), No. 800-1399-02, Sun Microsystems, Mountain View, Calif., Feb. 15, 1987, pp. 39-41.

19. *M88000 Architecture Specification,* Motorola Corp., 1986.

20. *i860 64-bit Microprocessor Programmer's Reference Manual* (V. 2), Intel Corp., Santa Clara, Calif, 1989.

21. *Intel 80960CA—Users Manual,* No. 270710-001, Intel Corp., 1989.

22. *IEEE Standard 1196-1987, Standard for a Simple 32-Bit Backplane Bus,* IEEE CS Press, 1988.

23. D.V. James, "An Audio/Video Simulation Facility," (presented at the 69th Convention of the Audio Engineering Society), preprint, AES, New York, 1981.

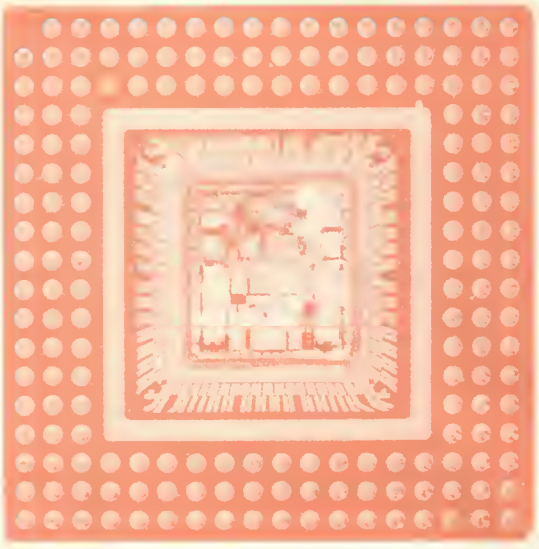
24. *ANSI/IEEE Standard 754-1985, Standard for Binary Floating-Point Arithmetic,* IEEE CS Press, 1985.

David V. James is a research scientist at Apple Computer and a major participant in the development of several IEEE bus standards. Prior to joining Apple, he worked at Hewlett-Packard, where he developed much of the I/O architecture for the HP PA-RISC family. His research interests include scalable interconnection architectures—from low-cost mouse interfaces to high-performance, massively parallel processors.

James holds the BS and MS degrees in electrical engineering and the PhD degree in electrical engineering/computer science from the Massachusetts Institute of Technology. He is a member of the ACM and the IEEE and chairs the IEEE P1212 CSR Architecture working group. He is also a member of the IEEE P896 Futurebus+, P1596 SCI, and P1394 Serial-bus working groups.

Direct any questions concerning this article to the author at M/S 76-2H, Apple Computer, Advanced Technology Group, 20450 Stevens Creek Blvd., Cupertino, CA 95014.

FEATURE



The 68040 Processor:
Part 2, Memory Design and Chip Verification

The second of a two-part series about the design of the 68040 discusses the memory subsystem, the external bus, chip and board testing, and design-verification methods.

Robin W. Edenfield
Michael G. Gallup
William B. Ledbetter, Jr.
Ralph C. McGarity
Eric E. Quintana
Russell A. Reininger

Motorola

The 68040 is a third-generation, full-32-bit microprocessor in the Motorola 68000 family. The 68040 integrates over 1.2 million transistors on one chip and is the first microprocessor to be fabricated with 0.8-micrometer CMOS (complementary metal-oxide semiconductor) design rules. It executes the complete 68020 microprocessor and 68882 floating-point coprocessor instruction sets. The sustained performance level is 20 VAX-equivalent MIPS (million instructions per second) and 3 Mflops (million floating-point operations per second) at a clock speed of 25 MHz. Pipelined integer and floating-point execution units that operate concurrently with separate internal memory controllers and an autonomous bus controller contribute to this performance level. Physical caches of 4 Kbytes each for instruction and data reside on chip. Separate address-translation caches of 64 entries apiece that operate in parallel with the instruction and data caches combine with each physical cache. This arrangement provides complete memory management in a virtual, demand-paged operating system.

In the February 1990 issue of *IEEE Micro*, we described design trade-offs, provided some insights into processor implementation, and detailed the integer and floating-point units. Here we discuss the memory subsystem, the external bus, chip and board testing, and design-verification methods.

Memory subsystem

The 68040 memory subsystem contains the instruction memory unit, the data memory unit, and the bus controller (see Figure 1). The subsystem provides instructions and data to the integer unit (IU) and the floating-point unit (FPU, not shown in figure) at a rate that matches the chip performance re-

0272-1732/90/0600-0022$01.00 © 1990 IEEE

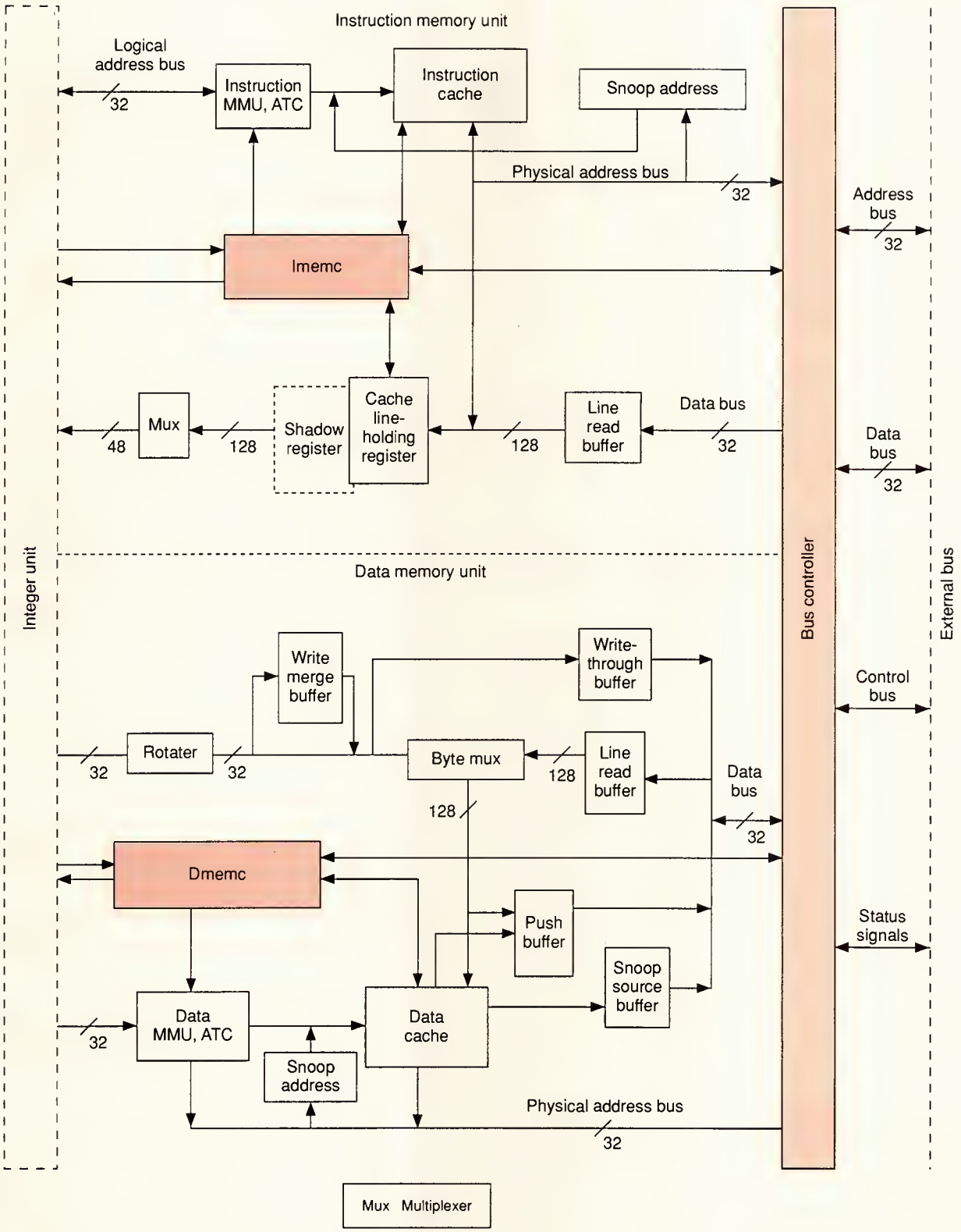


Figure 1. Memory subsystem and data paths.

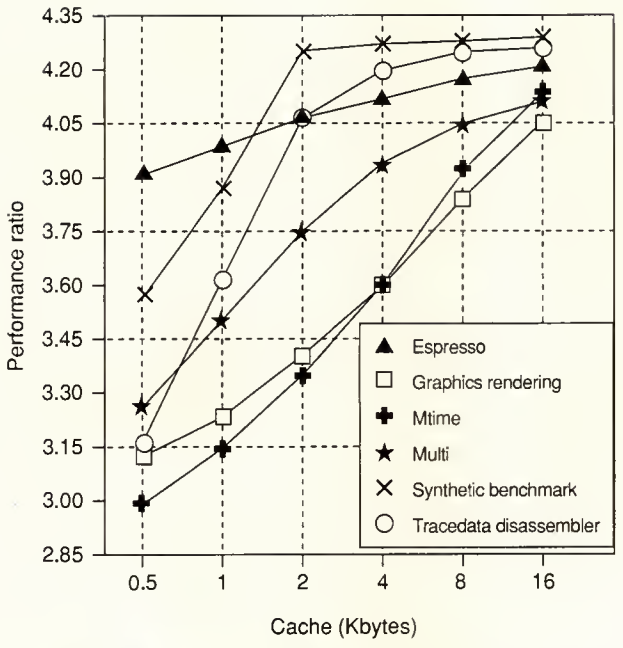


Figure 2. Processor performance relative to the 68020 versus cache size (where the 68020 equals 1).

quirements. Based on the data found in MacGregor, Mothersole, and Moyer[1] and on our analysis of trace data, we estimated that the IU would require transfer speeds of approximately 48 Mbytes per second from the instruction stream and 32 Mbytes/s from the data stream. Because we made the fundamental decision to have one external bus, we optimized the memory subsystem so that the majority of the required bandwidth comes from internal caches. The external bus handles cache load and unload functions. To this end, we chose a Harvard architecture memory subsystem with two memory management units (MMUs), two caches, and three separate controllers that function as follows:

- The data memory controller (Dmemc) maintains the data cache and its MMU.
- The instruction memory controller (Imemc) maintains the instruction cache and its MMU.
- The bus controller manages external bus protocol.

The three independent controllers and their associated buffers decouple external bus effects and cache management (loading or unloading) from instruction execution. Since the three controllers are independent, a request/acknowledge protocol handles communication between the two memory controllers and the IU or the bus controller. The IU requests data or instructions from the two memory controllers, and they provide data to service IU requests. The controllers also make requests of the bus controller that cannot be satisfied by the caches. Similarly, the external bus controller can request cache activity to satisfy requests from the external bus (snoops).

The autonomous controllers and buffers are essentially a pipelined memory system that isolates the IU from external bus activity, as Figure 1 shows. Figure 1 also illustrates the buffering between the IU and the memory subsystem as well as the buffering between the external bus controller and both internal memory controllers. We discuss this buffering later, in addition to the address-translation caches and cache line holding registers.

Internal caches

We chose the cache size and organization based on simulations performed on trace data with a cache simulator program. This simulator allowed us to run the collected instruction and data sequences with various cache sizes and organizations. Figure 2 shows some of the results of these simulations on various types of traces. These traces demonstrate a wide range of effectiveness provided by different cache sizes. From this data, we chose 4-Kbyte caches as a balance between performance requirements and silicon constraints. The hit rates that the simulator calculated from this data are consistent with those given by Smith.[2]

The data cache and the instruction cache are both four-way set-associative units with 64 sets of four entries (lines) each. Each cache line in each set contains 16 bytes of data for a total size of 4 Kbytes per cache. We chose the four-way set-associative organization to maximize hit rates in the caches while minimizing total silicon area. We preferred a fully associative cache to maximize hit rates. However, the amount of silicon required for a fully associative cache was prohibitive. We also considered direct-mapped and two-way associative caches, but the four-way set-associative organization provided superior performance without excessive silicon cost.[3,4] Results from the cache simulations performed on the trace data also supported this decision.

Since the memory subsystem provides logical-to-physical address translation by the MMUs, another fundamental design decision employed physical addressing for both caches. This approach makes the caches into mirrors of external, physical memory. Therefore, the operating system does not need to flush data or instructions from the caches on a process switch. Physical caches allowed us to provide external access to the caches (bus snooping) without requiring a reverse physical-to-logical address translation. Having physical caches also removed the problems associated with virtual address aliasing in the caches. Note that physical addressing into the caches and a 4-Kbyte minimum physical page size allow cache lookup concurrent with logical-to-physical address translation. These concurrent operations can occur due to the fact

that bits 11 through 0 are untranslated for a 4-Kbyte page. Cache lookup concurrent with address translation by dual MMUs guarantees that a cache lookup can complete in one clock cycle.

In addition to data or instructions, each entry in the caches contains an address (tag) and state information. This information consists of a validity bit in the instruction cache and a validity bit and four dirty (incoherent) bits in the data cache. Each dirty bit corresponds to a long word (32 bits) in the cache line. The data cache can handle writes to memory in two modes. In the write-through mode, each write to memory proceeds to external memory. The write also proceeds to the data cache when the address resides in that cache. In this mode, the Dmemc clears the dirty bits to indicate that the cache and external memory have the same data and are therefore coherent. In the copy-back mode, each write to memory proceeds only to the data cache. The appropriate dirty bit or bits are set to indicate that the cache line and external memory are incoherent (the cache line is dirty). The system writes (pushes) a dirty line to external memory when that entry location is allocated for another line of data.

We chose the line-validity method to reduce the number of bits required for state information. Line validity implies that external memory loads an entire line into the cache at once. Reading and loading an entire line benefits the instruction cache. Instruction accesses are largely sequential, and fetching the entire line is effectively an extended prefetch operation. The benefit in the data cache is less obvious because data accesses are not generally sequential. However, trace-data analysis indicated that loading the entire line into the data cache was advantageous. Further, we minimized the penalty for fetching an entire line of data by allowing cache accesses to occur in the processor concurrently with the external bus activity required for cache loading. We also designed the external bus protocol to use line-burst transfers from external memory to internal buffers without using internal cache buses.

Cache-maintenance activity refers to the line reads required to load the cache or to the pushes required to move a dirty line from the cache to external memory. When an access for the IU misses in the data cache—and the line to be replaced has dirty data—the Dmemc must perform two sets of actions. It must push the line currently in the cache and fetch the line containing the data requested by the IU from memory and load it into the cache. The Dmemc isolates the IU from cache maintenance primarily by buffering dirty lines (picked for replacement) in the push buffer. Thus the system can read the line before the push occurs on the external bus. This deferral of pushes minimizes the latency from a cache miss to data that is sent back to the IU, as explained later. The fact that cache accesses are allowed while a push is occurring on the external bus further minimizes the effects of the pushes.

The cache simulations that we performed indicated that 22.2 percent of all pushed lines had only 1 dirty long word, 3.8 percent had 2 dirty long words, 10.5 percent had 3 dirty long words, and 63.3 percent contained 4 (all) dirty long words. Because the additional status contained in each line indicates dirtiness for each long word, the Dmemc can push a line from the cache without requiring a full line burst on the external bus. When only 1 long word in the line is dirty, the Dmemc requests a long-word access to push the cache line.

Pushing 1 long word instead of an entire line saves a minimum of three external bus clock cycles. This savings increases when wait states occur on the external bus. The deferral of pushes, the overlap with cache accesses, and the variable sizes of pushes allow the processor to take maximum advantage of the copy-back mode while minimizing stalls and external bus bandwidth.

The internal Harvard architecture of the processor allows simultaneous accesses to the instruction and data caches. To save power—and since the size of all accesses into the caches is considerably less than 128 bits—the system does not read all 128 bits on every clock cycle. Instead, it reads one half, or 64 bits, of an accessed cache line. Consequently, the Dmemc requires two clock cycles to load a replaced dirty line into the push buffer. However, the Dmemc hides these two cache accesses under the external bus transfer during the line read.

Figure 3 on the next page demonstrates a hit in the data cache, which takes one clock cycle, followed by a miss in the data cache. When the miss occurs, the Dmemc picks a dirty line for replacement and loads it into the push buffer during the beginning of the external bus transfer. As shown in Figure 3, the Dmemc streams data received from memory immediately to the IU. During the remaining portion of the bus transfer, the IU can access the cache. Once the entire line is received from memory, the Dmemc loads the entire line of data (as well as both the tag and state information) in one cycle. The instruction and data controllers both stream data to the IU and allow concurrent bus and cache activity as well.

Although it is also feasible to load a cache as the data is retrieved from memory, we had two reasons for not choosing this type of partial load. Partial loads would have prohibited concurrent cache and bus activity, stalling the IU unnecessarily. Secondly, clean cache lines waiting for replacement remain in the cache until the bus cycle completes. This allows abnormally terminated bus cycles (through a cache inhibit operation or error from the bus) to avoid unnecessarily invalidating an entry in the cache. Delaying cache loading until the bus cycle completes and allowing IU access to the caches concurrently with bus activity reduces the penalty for loading the cache. At most, the penalty amounts to a denial of access to the cache for the one clock cycle required for the load.

In the 68000 family architecture, instruction length

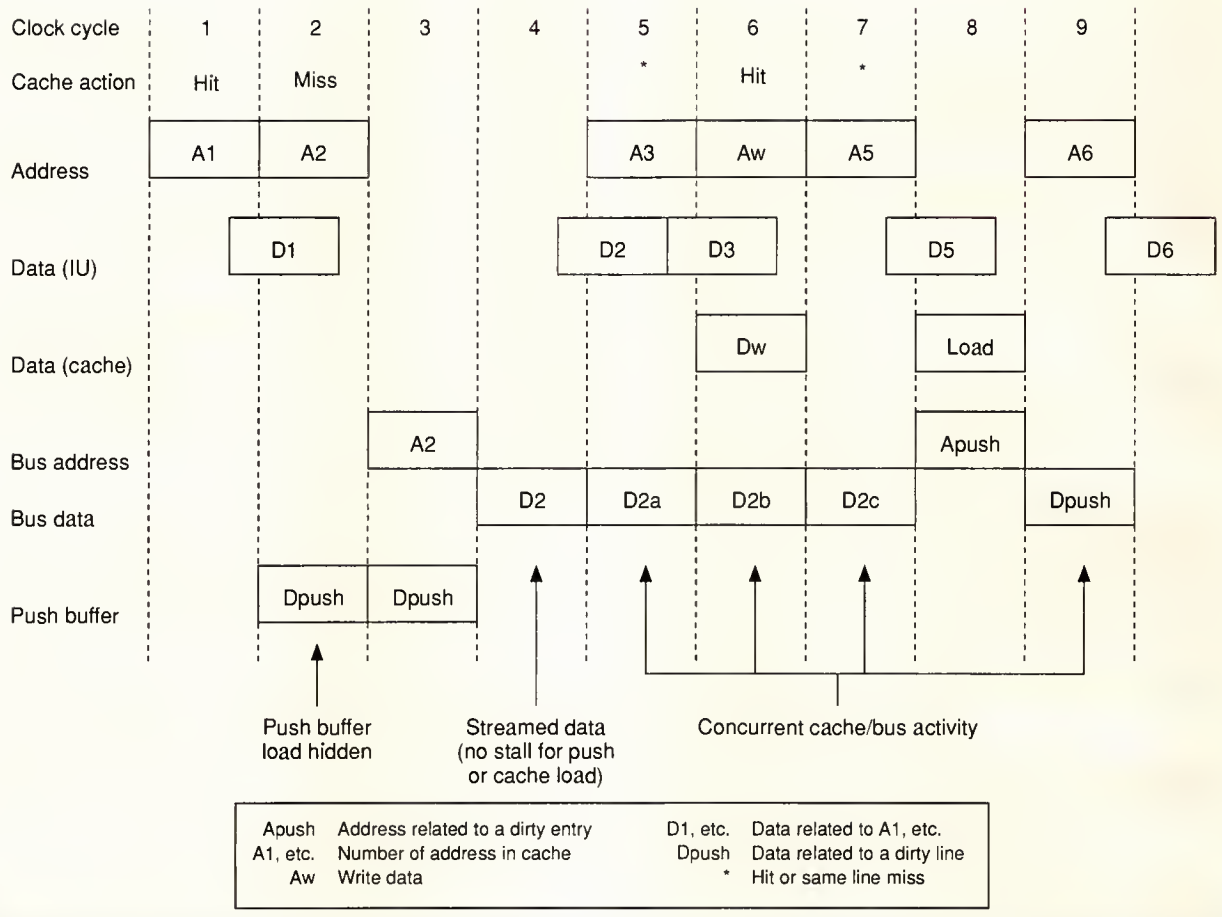


Figure 3. Data memory operation. Each column indicates a data read with overlap.

varies from 1 to 11 words. However, the length for optimized instructions does not exceed 48 bits (3 words). To make sure that the IU receives an optimized instruction every clock cycle, the Imemc maintains a 128-bit cache, line-holding register (CLHR) that is essentially a moving window on the instruction stream. The 48 bits required by the IU are multiplexed out of any word-aligned position in the CLHR. As soon as the last word in either half of the CLHR feeds into the instruction pipeline, the IU requests the Imemc to load that half of the CLHR with a new half line of instructions. This prefetching means that the CLHR does not hold a single cache line—but it can hold pieces of two cache lines.

Since the instruction cache can supply 64 bits in one clock cycle, the Imemc can fill the CLHR in advance of the IU requirement for the next instruction. To improve performance in the case of cache misses, the system loads each long word into the CLHR directly from the external bus as it is received. In addition, the Imemc forwards each long word received from the external bus to the IU based on the portion of the CLHR the IU requires. This streaming of instructions from the external bus to the IU minimizes the penalty for cache misses.

The system normally fills the CLHR sequentially. However, once a branch has been recognized by the program-counter calculation and decoding stage of the instruction pipeline, the IU requests the Imemc to fetch the target of the branch instruction and place it in the CLHR. The Imemc moves the current CLHR into a shadow register and fetches the target of the branch. If the branch is taken, the system employs the CLHR. Otherwise, it restores the shadow register to the CLHR, thereby avoiding refetching the not-taken path and minimizing branch penalties. When the not-taken path misses in the instruction cache, the Imemc requests the line from the external bus controller. When the instruction pipeline determines that the branch is not taken, further accesses can occur on the branch-taken path concurrently with the line read on the external bus—as long as they all hit in the cache. This "never mind" of

the cache line fetch minimizes the impact of a cache miss for a request down the wrong path of the branch.

On the data side, the variable operand length does not exceed 32 bits. When the IU requests a data operand, the Dmemc analyzes the alignment of the operand and decomposes the request into multiple cache lookups when required.

The fact that the data cache can supply 64 bits of data per clock cycle allows the Dmemc to service any read request that does not cross a half-line boundary in one clock cycle, regardless of alignment. Read hits that cross either a half-line or a line boundary require two clock cycles.

A read that misses requires one external bus transfer when the operand resides within a long word; otherwise, the read requires two external bus transfers. Misalignment within a line on a read miss does not require that the entire line be loaded before the data proceeds to the IU. The Dmemc feeds data to the IU as it is received from the external bus and multiplexes the proper portions of the necessary long words to the IU as well. With no-wait-state operation on the external bus, the IU only stalls two clock cycles on a cache miss for aligned data. The Dmemc informs the IU that its request is satisfied when it receives the last portion of data from the external bus. After the request is satisfied, the Dmemc accepts new requests from the IU. If in fact the next request is for another portion of the line coming from the external bus, the data again proceeds to the IU as soon as it is received from the external bus.

This streaming of data from the external bus directly into the IU helps both IU string operations and FPU double-precision operand loads. Also, if the next request can be satisfied by cache accesses alone, external bus activity does not stall the IU. In fact, the IU remains unaware of it. The Dmemc can service multiple IU requests before the line read completes and the cache line is loaded into the cache.

Once the line read completes, it takes one clock cycle to load the cache line into the cache. Consequently, the IU cannot access the cache for one cycle. This predicament stalls the IU when an operand read access is requested simultaneously with the completion of the line read. Writes do not stall the IU because they are passed off to the Dmemc and stored in the write buffer, thereby allowing the IU to proceed. Since the data cache can write an entire cache line at once, the Dmemc takes only one clock cycle to perform a copy-back write (of any alignment) into the cache, provided that the write operand does not cross a line boundary. When the Dmemc writes to a copy-back page and it misses in the cache, the controller requests a line read from the external bus. The line obtained is loaded into the cache and modified with the data written from the IU. This procedure is called write allocate.

The Dmemc accepts new requests for data accesses from the IU once the first data-transfer acknowledge signal is received from the external bus. Therefore, the IU does not stall waiting for the cache to be filled for the write miss. For writes to write-through pages, the Dmemc decomposes the request into aligned accesses to the external bus, which can (depending upon alignment) produce multiple accesses in the data cache as a side effect.

Memory management

The 68040's dual MMUs provide logical-to-physical address translation for instructions and data, respectively. Each MMU contains two transparent translation registers (TTRs) and an address translation cache (ATC) of 64 entries organized as four way set associative. Both the TTRs and the ATCs provide 32-bit, logical-to-physical address translation with a 2-bit extension controlled by the user. In addition to address translation, the MMUs provide supervisor and write protection and cache-moding information on a per-page basis. Supervisor-only instructions—including those that query and load an address and flush ATC entries—maintain the TTRs and ATCs. The ATC entries (either supervisor or user) are loaded by a microcoded table walk algorithm. This microcode uses separate supervisor and user root pointers as entry points to a three-level table structure of single-format, 4-byte descriptors. The ATC provides a 4- or 8-Kbyte selectable page size, and the TTRs provide a transparent mapping of segments from 16 Mbytes to 4 Gbytes in size.

The factors that motivated the design of the MMU were (in roughly this order):

- performance,
- ease of design,
- table memory usage,
- essential requirements,
- chip area, and
- 68030 compatibility.

Table-search algorithms. We analyzed several table-search methods, including hardware two- and three-level table searches and a software table search. Although the software table search method would have been the easiest to implement, we rejected it because it was too slow. Even with significant hardware assistance, a software table search was at least twice as slow as its hardware counterpart, reducing overall performance by 7 percent. Also, the software algorithms that could not use a 68040 hardware assist were 10 times slower than an equivalent hardware version. We felt that users would be unwilling to sacrifice this much performance.

We also evaluated two- and three-level table structures. A two-level structure would reduce the time required to perform a table search but would make operating system operations slower and less flexible

MMU segments

A typical operating system allocates each process its own private address space, which gives each process its own root pointer and table structure. The system allocates virtual memory to a process in units called segments. Segments can vary in size, and they grow or shrink as a process executes. Therefore, the system allocates an entire page table, rather than just a single page, for each segment. Marking a corresponding page descriptor valid or invalid and allocating or deallocating a physical page can add pages to, or delete them from, a segment.

The minimum amount of virtual space a segment claims is thus the amount claimed by a page table. A small segment size has several advantages.

First, the smaller segment requires less physical memory overhead for the associated page table. Less overhead means less waste of physical memory, faster paging of process information to and from the disk, and faster initialization of the tables.

Second, a smaller segment size allows a greater maximum number of segments to be allocated to a process. A trend in operating system design uses "lightweight" processes. One of these processes shares the logical address space of the parent process that spawns it, meaning that it receives one or more segments from the pool available from the parent process. If more segments are available, more lightweight processes can be created. Because these processes typically are small and of short duration, they avoid a great deal of process creation/deletion overhead, which improves performance.

These factors guided us in the selection of the 68040's three-level table structure. Assuming a 4-Kbyte page size, the 68040's structure uses a 7-7-6-12 logical address partitioning: the upper 7 bits of the logical address provide for the initial lookup, the next 7 bits for the second-level lookup, and the next 6 bits for the page-table lookup. The remaining bits are untranslated. Because the descriptors in the tables are all 4 bytes long, (2 ** 7 × 4 = 512) bytes are required for each first- and second-level table, and (2 ** 6 × 4 = 256) bytes are required for each page-level one. We compare this later to a 10-10-12, two-level, logical address partitioning. A 10-10-12 partitioning requires (2 ** 10 × 4 = 4 Kbytes) for each upper level and page-level table.

A typical Unix implementation allocates a minimum of four segments per process (text, stack, data, and miscellaneous). So, the 68040's table structure requires one first-level, one second-level, and four page-level tables. This adds up to 512 + 512 + (256 × 4) = 2 Kbytes of overhead for a minimum-size process. A 10-10-12 partitioning requires one upper level and four page-level descriptors, which add up to 4 Kbytes + (4K × 4) = 20 Kbytes of overhead for a minimum-size process. This amount of overhead is an order-of-magnitude more than that required by the 68040.

The 68040 7-7-6-12 partitioning provides a maximum of (2 ** 14 = 16K) segments per process, while 10-10-12 partitioning uses a maximum of (2 ** 10 = 1K) segments.

Consequently, the 68040's three-level structure results in less page table overhead and permits more lightweight processes to be attached to a parent process.

(see the box). Existing user opinion was split—some requested three levels and others two levels. We could have provided both via a mode bit, but this approach would slow down the table-search algorithm. In fact, we calculated that a three-level modeless table search would be as fast as a two-level moded search (with a three-level search as the second mode). In the face of this information, the performance benefits of a two-level table structure disappeared, so we chose a three-level one.

Page size. A performance requirement of the 68040 specified that the ATC and physical cache lookups must occur simultaneously in the instruction and data sides of the processor. The ATC uses the upper address bits, and the cache uses the lower ones. This requirement constrained the minimum page size because the same bits cannot be used for both lookups. In a four-way set-associative, 4-Kbyte physical cache, bits 9 through 0 of the logical address are ineligible for use in the ATC lookup. Thus the minimum possible page size was 2 ** 10 = 1 Kbyte. However, because future derivatives of the part could contain more cache, we chose a more realistic minimum of 4 Kbytes.

A survey of existing computer systems revealed a trend toward larger page size. Most existing systems use a 4-Kbyte page size, while the newer ones employ 8 Kbytes. Designers still debate the ideal page size. A large size increases the efficiency with which data can be paged to and from the disk and reduces the number of ATC faults. However, it also increases internal fragmentation (locations in a page not used by a process), thus increasing the likelihood that unused data is unnecessarily paged to and from the disk. We compromised and provided a mode bit that selects either a 4- or 8-Kbyte page size.

Set associativity versus full associativity. Using the collected trace data, we conducted simulations to determine the best associative organization for the ATCs. We estimated that a fully associative ATC would require twice as much chip area as an equivalent four-way set-associative ATC. Figure 4 compares the two implementations for various ATC sizes and page sizes. The hit rate is essentially the same for 32-entry, fully associative and 64-entry, four-way set-associative ATCs. We chose the latter organization because

1) it provided a minor improvement in hit rate;
2) the same memory cell could function in both the ATC and in the physical caches, which reduced the design effort; and
3) a 64-entry ATC fares better on those rare occasions when users run a program that has a very large working set. (However, the programs on which we collected trace data did not come from this category.)

Note that in Figure 4 the point chosen (at 64 entries) is well past the knee of the curve. This selection provides a margin of safety for instances in which the 68040 operates in an environment that requires significantly more entries than those traced.

As the chip's design progressed, we periodically reviewed the ATC size. We found that the chip size would not be affected if the ATC size was reduced, due to the way other blocks were placed around the ATC.

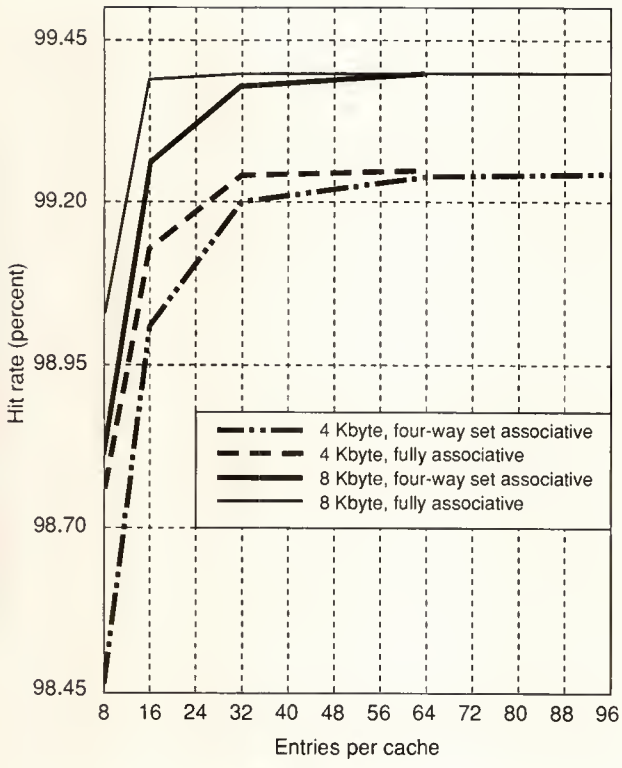


Figure 4. Cumulative ATC.

Compatibility. Performance goals made the 68040 MMUs inherently incompatible with the 68030. Fortunately, all MMU incompatibilities are confined to the operating system, leaving user code completely compatible. Some fundamental incompatibilities exist in the following 68040 characteristics:

- a minimum page size of 4 Kbytes,
- cache-mode selection on a page basis,
- obsolescence of the 8-byte descriptor due to the larger page size, and
- a global bit.

In addition, our desire to improve the performance of table searches (discussed later) caused some incompatibilities. Nevertheless, we attempted to remain as compatible as possible. In fact, when an 68030 operating system uses only 4-byte descriptors and has the same three-level table structure as the 68040, the two are compatible except for the 68040 features just described.

Other issues. Since many operating systems require that supervisor accesses be mapped differently from user accesses, we provided separate user and supervisor root pointers. The appropriate root pointer selection depends on the state of the supervisor bit of the status register. We rejected an alternative method that would consider the supervisor bit a thirty-third bit of the logical address and alter the table structure so that the first-level table is 1 Kbyte instead of 512 bytes in size. This approach simplifies the implementation but has the disadvantage of distributing the first-level supervisor table among all processes. It also increases the overhead of changing a supervisor mapping. In addition, it precludes the possibility of paging to disk the first-level table of a process (since the table must be updated when a supervisor mapping changes).

When processes execute, they use common operating system facilities. Supervisor calls access some of these facilities; other facilities simply consist of common subroutines accessed in user space. During execution of a particular process, the ATC contains some mappings that are process specific and other mappings that are common to all processes. The user-specific mappings must be flushed when the execution of one process ends and the next one begins. However, performance improves when the global mappings can remain (since the table searches for these areas do not recur). We added support for this capability by providing a global bit in the page descriptors. This bit selectively flushes entries during ATC flushes. The PFLUSHAN (processor flush all nonglobal) instruction removes only nonglobal entries.

Implementation issues. To reduce the amount of silicon devoted to memory management (based upon the good results in the design of the 68030), we decided table searches should be performed in microcode by the IU. (The IU executes instructions.) This decision did indeed reduce the silicon area required, but it also added numerous and difficult-to-debug boundary conditions. Another complication consisted of the reduction in performance of table searches caused by instruction-stream accesses. These searches must wait for the instruction pipeline to clear (flatten). Flattening the instruction pipeline hurts performance because it creates bubbles or time slots in which no useful activity can take place.

However, we used an implementation technique to increase table-search performance while imposing a negligible increase in silicon area and design complexity. This technique increased the speed of table searches by carefully defining the table descriptors and relying on certain situations only present during table searches.

Normal table-search sequences require three accesses to memory. These accesses cannot occur in a back-to-back fashion because the addresses for the second and third accesses are derived in part from information contained in the descriptor obtained by the prior access. Consequently, we tried to allow a maximum of one dead clock cycle between each access. However, the table-search microcode must accommodate the fact that an upper level descriptor can indicate an invalid mapping. This condition must inhibit further accesses.

An obvious solution to this problem is to slow down table searches. The approach used on the 68030 was to first branch in microcode on the descriptor information and only retrieve the next level when conditions are correct. Instead, the 68040 microcode simultaneously branches on the descriptor information and also requests that the next level be retrieved, relying on the data memory unit to ignore the access if bit 1 of the address is not 0 during a table search.

During a table search, all accesses are 32 bits long and aligned. Thus address bit 1 should normally be 0. The resident bit of a descriptor is only 0 when the descriptor is invalid. So—in the same cycle in which the new address is formed—the resident bit of the descriptor is inverted and gated into bit 1 of the newly formed address. When the microcode branches on bit 1, it relies on the fact that the requested access was ignored.

External bus

This bus provides transfer of cache line data between external memory and the internal caches. We organized these caches to minimize separate bus accesses. We widened the cache line to accommodate more than the natural data requirements of either the instruction pipeline or the execution units. We then tuned the external bus and on-chip controller for the efficient transfer of entire cache lines.

The external bus uses a synchronous protocol for the transfer of 8, 16, or 32 bits in a minimum of two clock cycles. The bus transfers 128 bits in a minimum of five clock cycles during a burst transfer. We chose a synchronous bus over the more familiar asynchronous buses found on the 68020 and 68030 designs. We wanted to allow the external bus to scale to higher speeds without making the timing specifications impossible to design within real systems. The synchronous design also allowed for the highest performance interface between the internal bus requesters and external memory in that a cache miss incurs no synchronization delays.

The design supports I/O devices that can transfer bytes, words, and long words on the external bus. In departure from the 68020 and 68030, the external bus does not support dynamic bus sizing. Certainly the vast majority of transfers in high-performance systems based on these earlier microprocessors do not use dynamic bus sizing. We felt that the burden in silicon, external bus performance, and design effort far outweighed the convenience provided to the external system design. In addition, because we considered I/O devices to be part of the operating system environment, the lack of complete compatibility at the bus transfer level was not as important as the performance and design schedule goals.

Protocol. Data transfers on the external bus in the following sequence:

1) The processor drives address bits and associated attributes like transfer size and read/writes. It also asserts a transfer-start (TS) signal to indicate that the address bus is valid. In a departure from previous 68000 family members, the 68040 TS does not indicate bus validity at the assertion point, but rather that the bus is valid in a particular clock cycle.

2) Within the following clock cycle (at the soonest time), the selected slave responds by asserting a transfer-acknowledge (TA) signal and either drives read data onto the data bus or accepts the write data presented by the processor. A slave can control the transfer rate. If TA is not asserted, the processor simply waits.

3) On byte, word, or long-word transfers, the processor accepts only one TA per transfer. Cache line transfers require four TA assertions on valid clock edges (this is the definition of a line burst). On writes, the processor drives new data onto the 32-bit data bus during the clock cycle after each assertion of a TA. On reads, the external memory must place new data on the bus during the clock cycle after it asserts TA.

When a bus error occurs, the responding slave asserts the transfer-error-acknowledge (TEA) signal instead of

a TA. The bus controller design allows a TA or TEA to be asserted on the bus as late as the arrival of data, thereby increasing the effective address-to-response time considerably over previous-generation 68000 family designs.

A bus slave can retry transfers by asserting both a TA and TEA at the same time to the processor. To simplify internal boundary conditions associated with retry and bus-relinquish situations with snooping, the protocol only allows a retry on the first long-word transfer in a burst.

The protocol accommodates nonbursting, 32-bit-wide memory as a sort of "macro," bus-sizing, transfer mode. At the termination of the first long word on the bus, the bus slave responds either by accepting write data or by providing the read data. Along with the TA, the slave asserts a transfer-burst-inhibit pin to indicate that the processor must enter the "fake" burst mode to transfer the remaining 3 long words. The bus controller then issues three separate long-word transfers, rolling the addresses appropriately. This option is useful for external ROM since accesses to ROM are not usually required to operate at full bus performance. Because an entire line is transferred, the data enters into the appropriate cache.

Arbitration. Current members of the 68000 family use an on-chip arbiter for the external bus.[5] This arbiter allows an external bus master to request the bus from the processor at any time but only grants the bus ownership when a read-modify-write (RMW) sequence is not running on the bus. RMW sequences occur when instructions need to access more than one operand and prevent other bus masters from interrupting those accesses. The other 68000 family members prevent another bus master from using the bus while an RMW is in progress.

As we talked to users, we found that some system designs need to allow alternate bus masters to use the bus in the middle of one of the "indivisible" RMW sequences. A general problem occurs when a system has both on-board RAM and a processor, as well as an off-board bus to other system memory. Other processors or DMA devices on the backplane bus can access on-board RAM as part of the address space of the overall system memory. A deadlock can occur when the on-board processor begins an RMW transfer sequence to on-board memory just as an off-board processor or a DMA device accesses the same memory. Although the system designer can work around the deadlock problem, the known procedures increase the system cost considerably.

The external bus-arbitration scheme allows more flexibility in the design of board-level arbitration. We decided to move the arbitration unit off chip to allow the system designer to choose the arbitration scheme. Simply removing the bus-granting signal to the processor can interrupt locked sequences at any point. The procedure requires alternate bus-master transfers that do not involve an access to data that is the object of a locked sequence. A system designer can allow those transfers to be the highest priority on the bus to obtain the absolute minimum bus latency. Consequently, the external bus arbiter must be aware of locked sequences. The arbiter must also prevent a locked sequence from being interrupted when

- the interruption is to service another bus master, and
- the bus master accesses the same data that is the object of the locked sequence.

Snooping. The external bus supports a uniprocessor snoop protocol to both on-chip caches. The protocol allows high-performance, off-chip DMA to efficiently use the bus in uniprocessor environments with the data cache in copy-back mode. The protocol does not directly support multiprocessor environments because we chose to maximize uniprocessor performance. This protocol supports both reading from and writing to on-chip cache data and is controllable on a per-transfer basis. The data cache can be updated on snoop writes and supplies data to the snoop master on reads (when a line is dirty). The protocol also allows the instruction cache to be invalidated on writes.

Each bus transfer by an alternate master indicates on the bus (via the pins) what kind of snooping should be performed by the processor: inhibited, read source, read source with invalidate, or write to data cache with invalidate in instruction cache. In general, each bus access that is marked "snoopable" costs one or two cache lookups into the data cache (depending on the bus transfer size) and zero or one lookups into the instruction cache. A snoop lookup in either cache may cause a stall in the IU or the FPU, depending on internal activity. Note that on noninvalidating reads the instruction cache is not accessed because it does not have dirty data.

One can think of the read source with invalidate snoop type as a distributed cache push operation in which the push happens on demand from the external bus instead of from internal sources. Users indicated that the effective use of large, on-chip caches with DMA operations requires something faster than the serial execution of cache management instructions in terms of overall system throughput. This snoop type gives system designers a fast way to move data from memory (either in the on-chip caches or in external memory) to the disk (page-out sequence). When a DMA controller performs line reads to move the data to a disk controller (page-out), it marks the snoop type as read source with invalidate. The processor then accesses both caches. The access to the instruction cache invalidates any line within the cache. The access to the data cache requires it to load the snoop source buffer and then invalidate any line within itself. The bus

controller parcels this line to the DMA controller from the snoop source buffer, effectively moving the line to memory. This invalidation procedure means that the DMA controller can mark bus transfers nonsnoopable during block moves or page-ins from the disk to memory because the lines are not in either cache. This minimizes the effect that snooping has during a page-in, page-out sequence on instruction execution.

We decided not to provide duplicate cache tags because we considered snoop performance less important than raw instruction execution. Devoting silicon area to a duplicate tag structure for snoops would reduce the area available for on-chip cache by approximately 30 percent.

Timing specifications. The processor has two separate input clocks: the bus (Bclk) and the processor (Pclk). All external bus timing refers to Bclk, while internal timing derives from Pclk. (However, all internal operations cycle at the Bclk rate.) Bclk must be exactly half the frequency of Pclk. Internally, Pclk provides timing reference points within each Bclk period. A phase-locked loop provides a close internal relationship between the internally derived timing based on Pclk and Bclk. The external bus uses Bclk to provide synchronous timing for bus transfers.

Early in the design process, we looked at the advantages of using asynchronous clocks. We rejected this approach due to the internal synchronization problems between the data sources, destinations, and external timing. One of the initial design goals was to provide a bus design and protocol that would minimize the latency from a cache miss to available data in the IU. We saw synchronous timing between the bus controller and the IU as the best way to accomplish this goal.

The compatibility of the external bus to existing TTL (transistor-transistor logic) appeared to be an important requirement for this generation of microprocessor. We anticipate advances in TTL technology that will allow zero-wait-state buses to be built for high-performance microprocessors. Users indicated that system cost would require most logic to remain TTL-compatible in the near future.

In prior Motorola microprocessors,[5] all timing specifications assume that output buffers are driving a lumped capacitive load. With the external bus clock speed increasing to 25 MHz and beyond, many users have had difficulties designing systems using the lumped-load model because it is not an accurate representation for real designs. This problem occurs primarily because the output buffer source impedance must decrease to drive a capacitive load faster. With a lower output impedance, the transmission-line effects of a fast output buffer become more evident in such system problems as line ringing and induced noise.[6]

Data, address, and control lines in the 68040 are individually selectable by group. They can drive either an unterminated, controlled-impedance transmission line or a terminated transmission line in the highest performance systems. The unterminated buffer drives a 30-ohm loaded bus with minimal ringing (source terminated). The terminated buffer has a low output impedance designed to drive a 50-ohm, DC or AC, terminated transmission line. With this environment, the fast rise and fall times of the output buffers result in high-performance bus designs without the problems associated with impedance mismatches.

The MOVE16 instruction

Since the 68040 provides fast on-chip, floating-point operations and integral memory management, we did not include the 68020 coprocessor protocol in the design. However, the processor still must move data and instructions between main memory and external system hardware such as a graphics engine. Consequently, we decided to include a method to provide higher performance than was available with the coprocessor interface on previous designs.

We had already added the hardware on chip to buffer cache line refills and pushes to minimize the effects of cache misses and the write-back of dirty cache data. We next examined the capability of this hardware to support a memory controller instruction in the instruction set to move a cache line between two memory addresses. This investigation resulted in the MOVE16 instruction. This simple instruction moves a cache line from one address—which may reside in the data cache—to another address outside the cache. The instruction is not a true data type since the IU cannot operate on the data, but MOVE16 does allow the processor to move data quickly on the external bus. It simply loads the line buffer in the on-chip data memory unit with the read data and writes it to a different address. The burst transfer type on the bus sustains the highest performance transfer of a cache line between two memory addresses that this bus can provide.

We compared this instruction with existing instructions. Our performance model predicted that an unrolled loop of four MOVE.L instructions (memory to memory) would sustain one half the transfer rate of a simple MOVE16 loop. The actual chip demonstrated this performance improvement. The execution pipeline can hide most of the loop overhead on top of the external bus transfers to provide this performance. The MOVE16 instruction can provide programmers with nearly all of the available external bus bandwidth, allowing low-cost system designs to perform DMA-like operations without the cost of a separate DMA controller.

Chip and board test

We designed the processor with testability in mind

for both the user and the factory. For the user, the processor includes dedicated test logic that is fully compliant with the *IEEE 1149.1 Standard for Test Access Port and Boundary-Scan Architecture*.[7] Problems with testing high-density circuit boards led to the development of this standard under the sponsorship of the IEEE Test Technology Committee and the Joint Test Action Group (JTAG). The 68040 supports circuit-board-level test strategies based on this standard. For factory testing, the processor employs structured design techniques for random logic and special test modes for embedded arrays. We tested the execution data paths by using functional test vectors. We believe that this mix of design-for-test techniques provides a good trade-off for circuit area overhead and chip testability. The user and factory testing logic are completely independent. No interfaces to the factory testing logic exist through the IEEE 1149.1-based port.

IEEE 1149.1. The test access port (TAP) and boundary-scan interface include the four required test pins TMS, TDI, TDO, and TCK, and the reset pin TRST. The boundary-scan register consists of a shift register that provides access to all chip clock and signal pins along with associated control signals. The implementation includes the three mandatory instructions BYPASS, SAMPLE/PRELOAD, and EXTEST, but does not include the optional public instructions. The latter instructions provide additional test capabilities.

Five unique instructions can be loaded into the 3-bit instruction register: BYPASS, EXTEST, SAMPLE/PRELOAD, BYPASS2, AND EXTESTZ. The BYPASS instruction selects a bypass register and forms a single-bit shift-register path from the serial test data input pin TDI to the output pin TDO. This configuration allows shifting through the bypass register when testing other devices that share the common serial input line. The EXTEST instruction selects the boundary-scan register. By using the TAP controller, the register can scan user-defined values into the output buffers and capture values presented to input pins. It can also control the direction of bidirectional pins and enable the output drive of three-statable output pins. The SAMPLE/PRELOAD instruction provides a means to capture a snapshot of system I/O signals and to initialize the boundary-scan register prior to the selection of the EXTEST instruction. The EXTESTZ instruction disables the drive on output-only pins, and the BYPASS2 instruction provides a bypass mode without clocking restrictions.

Factory testing functions. As mentioned, the processor uses three different design-for-test techniques for addressing the issues of factory testing. We analyzed trade-offs for each area of the design to see which test technique would best fit each section.

We picked a functional style for the data-path elements for historical reasons. A great deal of functional tests existed from previous designs. For the random logic and ROM/PLA (programmable logic array) areas, we employed a structural style of design for test. We used existing latch structures to form scan chains, which allows automatic generation of test patterns for these structural areas. For the remaining parts of the processor—specifically the cache and cache tags—special test modes came into play. These test modes allow the caches and tags to be accessed like RAM structures from the pins of the processor. In this way, pipelined one-cycle accesses are available to the caches and tags. This capability allows the use of any type of memory test. In the design of these test modes, we determined that no one or two types of memory test would meet all the needs of factory test. Consequently, we developed a general interface to the internal memory circuits.

Design and verification

We designed the processor in a top-down fashion. From the user specification, we divided the chip into three blocks: the IU, the FPU, and the memory subsystem. We subdivided the three sections into control stores, random logic, and data-path elements for the IU and FPU, data and instruction cache/tag memory, ATCs, internal memory controllers, and the external bus controller. Our modular approach divided the design into pieces that exhibited minimum interaction with each other.

Once the components of the processor were decided and the interfaces were worked out, the process of high-level modeling began. We chose Verilog (verilog_ref)[8] for the high-level models. These models provided design specifications for each section of the chip, which totaled approximately 36 separate sections. The style of the modules mixed register-transfer-language and functional description types of approaches. The RTL style dominated the data-path sections, while a functional style prevailed in the controller sections.

After each of the behavioral models was debugged, we performed the process of implementation. For the data-path sections, a simple transformation of the RTL style model served as a register data path. The more random areas required a more automated logic synthesis approach. We relied on an internal tool to extract verbatim the logic specified in the behavioral models for the Dmemc and the bus controller. This extracted logic was then fed through Espresso[9] and MisII[10] from the Berkeley Oct tool set to produce layout-ready transistors. These devices were then synthesized directly into layout.

We relied on a hierarchical method of design verification. This choice came from the observations that each section needed different verification requirements and that the sections needed to be debugged independently. Also, we wanted to evaluate design changes and design error fixes as quickly as possible. Two methods

provided early verification of the modules: instruction sequences for the IU and FPU and dedicated stimuli for other modules.

The dedicated stimulus for the memory subsystem emulated the boundary interfaces so that each section could be debugged separately. This low-level verification then extended to levels in which several memory elements were placed together. We used this method for all three memory subsystem controllers.

We used instruction sequences for stimuli in the IU and FPU stand-alone sections. Early simulation–as in the memory section–allowed us to evaluate design changes. At the last stage of the verification process, we simulated the entire design with these same instruction and data sequences.

This hierarchical verification environment also provided a framework for the verification of the structural, or gate-level, description. Simply replacing a behavior section with a gate representation allows both descriptions of the design to be verified with the same stimulus. Thus we could verify each of the gate-level representations at larger and larger configurations.

We discussed the memory subsystem, the external bus, chip and board testing, and design-verification methods in the 68040. We concentrated on the trade-offs we made to give the 68040 greater parallelism, fewer dead cycles, and consequently better performance than any other comparable microprocessor.

Acknowledgments

We thank all of the people who have worked on the 68040 design and have helped to make its implementation possible.

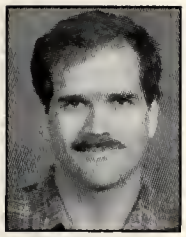
References

1. D. MacGregor, D. Mothersole, and B. Moyer, "The Motorola MC68020," *IEEE Micro*, Vol. 4, No. 4, Aug. 1984, pp. 101-118.
2. A.J. Smith, "Cache Evaluation and the Impact of Workload Choice," *Proc. 12th Int'l Symp. Computer Architecture*, IEEE Computer Society Press (microfiche), Los Alamitos, Calif., 1985, pp. 64-75.
3. C. Alexander et al., "Cache Memory Performance in a Unix Environment," *Computer Architecture News*, Vol. 14, No. 3, June 1986, pp. 41-70.
4. J.E. Smith and J.R. Goodman, "Instruction Cache Replacement Policies and Organizations," *IEEE Trans. Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 234-241.
5. Motorola, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, N.J.
6. W.R. Blood, Jr., *MECL System Design Handbook*, Motorola, Phoenix, Ariz., 1988.
7. *IEEE Standard 1149.1, Standard for Test Access Port and Boundary-Scan Architecture*, IEEE Press, Piscataway, N.J., 1990.
8. P. Johnson, "Mixed-Level Design Tools Enhance Top-Down Design," *Computer Design*, Vol. 29, No. 1, Jan. 1989, pp. 85-88.
9. R.K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Hingham, Mass., 1984.
10. R.K. Brayton et al., "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, No. 6, Nov. 1987, pp. 1062-1081.

Edenfield

Gallup

Ledbetter



McGarity

Quintana

Reininger

Robin W. Edenfield is a principal staff engineer at Motorola and was primarily responsible for system design of the cache and memory management controllers for the 68040. He previously worked in the design, test, and production of microprocessor systems at several other companies. His current interests include microprocessor architecture and memory subsystems.

Edenfield received the BSEE degree from the Georgia Institute of Technology, where he was elected to Eta Kappa Nu.

Michael G. Gallup, with the microprocessor group at Motorola since 1981, is a circuit design engineer and a member of the technical staff for the high-end design group. His areas of interest include semiconductor devices, digital systems, and computer programming.

Gallup received the BSEE and MSEE degrees from the University of Nebraska at Lincoln. He is a member of the IEEE Computer and Circuits and Systems Societies.

William B. Ledbetter, Jr., is a principal staff engineer in high-end design at Motorola. He was responsible for the external bus specifications and protocol—as well as the system design of the bus controller—for the 68040. He also was a member of the architecture design team for internal memory. He previously designed graphics systems for Data General.

Ledbetter received the BSEE and MSEE degrees from Texas A&M University. He is a member of the IEEE Computer Society.

Ralph C. McGarity is a principal staff engineer and was the systems technical project leader for the 68040 project. He has spent seven years at Motorola designing 68000 family parts. His interests include microprocessor architecture design, implementation, and verification.

McGarity received the BSEE degree from Rice University and the MSEE degree from Carnegie Mellon University. He is a member of the IEEE, ACM SIGarch, Phi Beta Kappa, and Tau Beta Pi.

Eric E. Quintana has been with the microprocessor group at Motorola since 1984 as a staff engineer and is the architect for the floating-point section of the 68040. He previously worked as a design engineer on the 68020 microprocessor and later on the 68882 floating-point coprocessor.

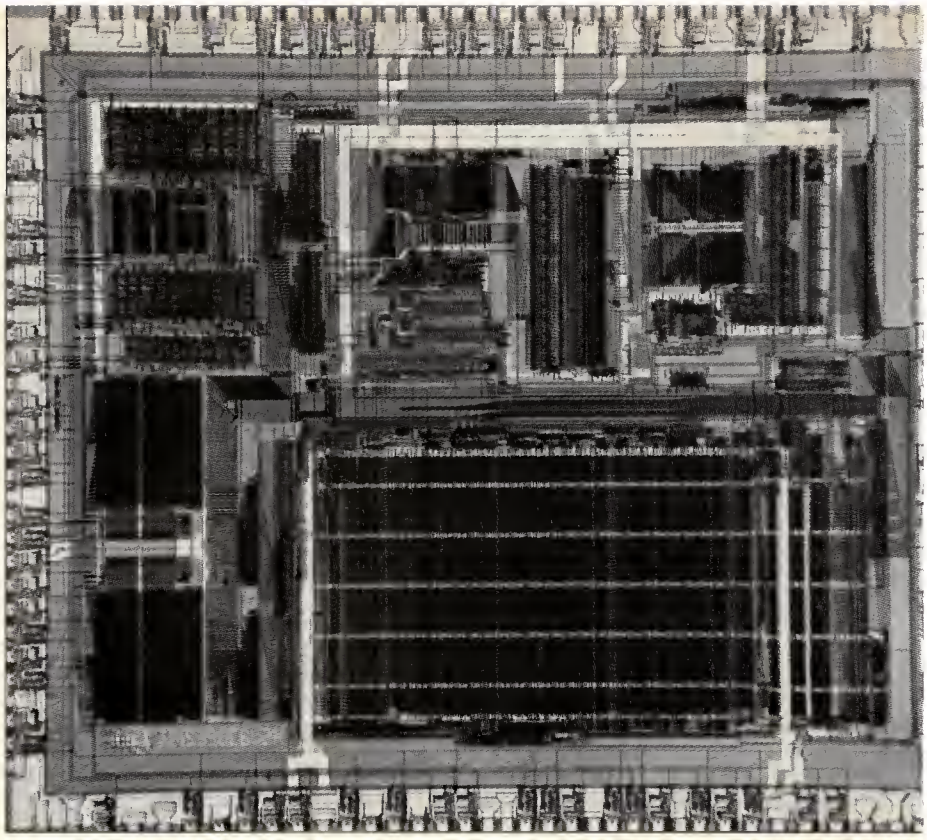
Quintana received the BSEE degree from Texas A&M University.

Russell A. Reininger is a principal staff engineer for high-end microprocessors at Motorola. He was responsible for the system design of the integer unit in the 68040. Before joining Motorola, he worked for six years at Tracor Aerospace in Austin, Texas, where he designed radiation-hardened military computers.

Russell received the BSEE degree from the University of Texas at Austin.

Readers can direct questions regarding this article to Ralph McGarity, Motorola Inc., M/S OE37, 6501 William Cannon Drive West, Austin, TX 78735-8598.

FEATURE



The TMS390C602A Floating-Point Coprocessor for Sparc Systems

Merrick Darley
Bill Kronlage
David Bural
Bob Churchill
David Pulling
Paul Wang

Texas Instruments Incorporated

Rick Iwamoto
Larry Yang

Sun Microsystems, Inc.

The Sparc architecture developed by Sun Microsystems seeks to improve the computational capability of workstations by using RISC (reduced instruction-set computer) techniques to process data faster than conventional CISC (complex instruction-set computer) computers.

Sun licenses Sparc (Scalable Processor Architecture) to semiconductor manufacturers that are free to add their own features and process technology and sell at their own prices. However, a common set of RISC specifications support all Sparc implementations.

One recent Sparc processor is a two-chip configuration supplied by Texas Instruments. These devices include the TMS390C601 integer unit (IU) and the TMS390C602A floating-point unit (FPU). Here, we discuss the second device, a highly innovative coprocessor, which lets the processor execute single- or double-precision floating-point instructions concurrently with IU operations.

0272-1732/90/0600-0036$01.00 © 1990 IEEE

We installed dedicated floating-point hardware in the FPU to increase the performance of the system. Running at clock periods as small as 20 nanoseconds, the chip should deliver 5.5 million double-precision floating-point operations per second under the Linpack benchmark (50-MHz clock rate). The FPU provides single- and double-precision arithmetic functions of addition, subtraction, multiplication, division, square root, compare, and convert.

We used TI's EPIC-IAE semiconductor processing technology, a 1-micrometer process with transistor gates shrunk to 0.8 μm, to build the FPU. The static CMOS (complementary metal-oxide semiconductor) design makes use of pass-logic gates that have single-phase clocking and extremely low standby power. We packaged the chip in a 143-lead, ceramic pin grid array.

Floating point and RISCs

Engineering workstations usually execute complex calculations that involve extensive number-crunching, often of numbers (data) in floating-point format. Most RISC architectures, therefore, support floating-point data in their instruction sets, and the 601/602A chips are no exception. The 602A FPU, with appropriate system software, provides Sparc-compatible floating-point arithmetic in accordance with the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*.[1]

This standard does not stress simplicity like the RISC methodology; instead, it emphasizes mathematical quality. A full hardware implementation requires significantly more complexity than other floating-point standards. For example, the IEEE Std.-754 requirement to round off a calculation, as if the operation were done to infinite precision, calls for a considerable amount of additional multiplier hardware.

A large segment of the workstation market clearly needs high-performance floating-point capability. Given this need, the benefits of IEEE Std. 754 quickly secured its adoption instead of the alternative of defining a floating-point instruction set and format for a given RISC architecture.

The design problem becomes one of reconciling complexity of a floating-point instruction set and the simplicity of a RISC processor. Designers solved the problem by not directly burdening the RISC CPU design with the complex floating-point instructions. Instead, they either implement the instructions in software in a true RISC design or provide a floating-point coprocessor.

The CPU implementations offered today are a cross between these two approaches. A compromise design lets a RISC floating-point coprocessor handle the most frequent floating-point instructions, while software traps accommodate very infrequent instructions or infrequent operands. We chose to embody this kind of

A dedicated floating-point coprocessor provides Sparc architecture implementations with high-performance IEEE Std.-754 mathematics. These operations include 15 cycles of register-to-register latency for double-precision divides and 18 cycles for double-precision square roots.

FPU architecture to accompany our Sparc RISC IU.

Our FPU integrates two predecessor chips onto one chip. TI had designed one of these predecessors, the ACT8847, as a general-purpose floating-point processor to support IEEE Std. 754.

Sun designed the second chip, the CY7C608, to control the ACT8847 and provide the required interface for the IU. To accomplish the design of these chips, the separate design teams operated independently at different sites but communicated frequently. A separate TI and Sun design team accomplished the FPU integration.

A top-level view

A typical Sparc processor, as seen in Figure 1, contains an IU, an FPU, and a memory system. All this is interconnected through a dedicated two-bus (32-bit address, 32-bit data/control) structure.

The IU directs the operation of the FPU by controlling the issuance of instructions and responding to both integer and floating-point traps. It also generates addresses for floating-point loads and stores, and performs branch instructions based on the floating-point condition codes generated by floating-point compare instructions.

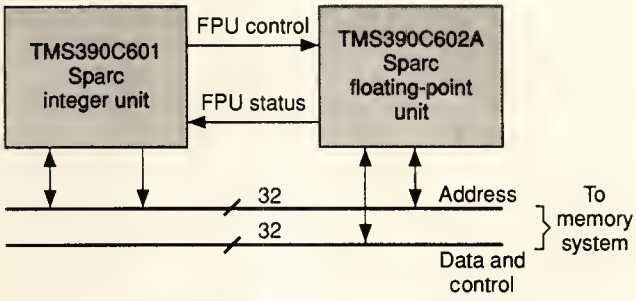


Figure 1. In Texas Instruments' version of a Sparc processor for RISC-based workstations, an integer chip (TMS390C601) works with a floating-point chip (TMS390C602A) to provide the complete Sparc function.

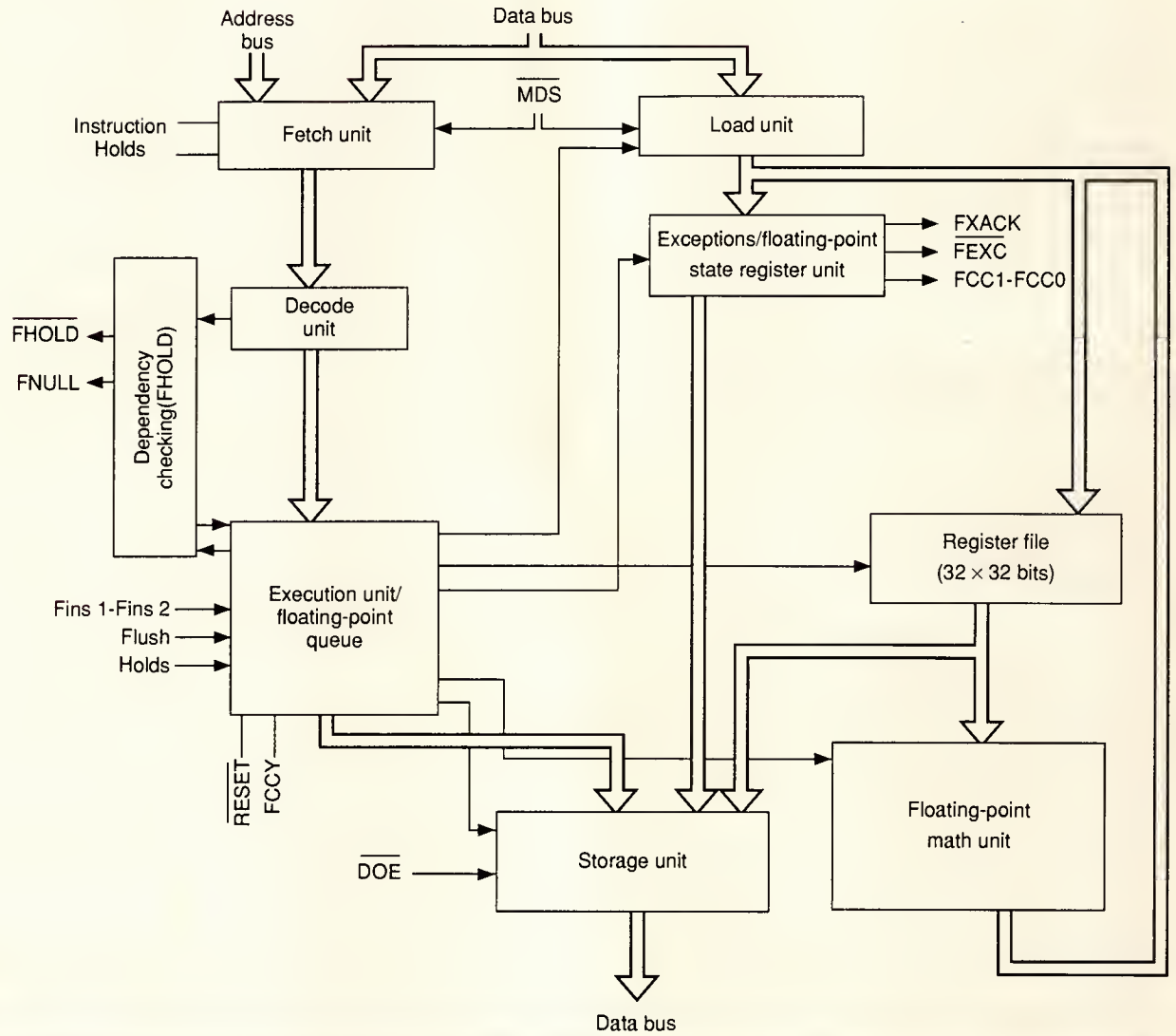


Figure 2. Operating under the control of its IU, the FPU uses its multiple functional blocks to execute load and store instructions and operate instructions.

The FPU can be divided into functional blocks (Figure 2). The data path consists of a math unit to execute the required arithmetic; a register file to store operands and results; an exceptions/floating-point state register for storing the rounding modes, exception masks, and arithmetic status; and a load unit to load the register file and floating-point state register from the data bus. The control path consists of an instruction/address queue to hold pending instructions and their addresses, a decode unit for decoding instructions, a dependency checking unit, and a fetch unit. Common to both paths is a store unit to output data and status during normal execution and instructions and addresses during exception handling.

Inside the queue

The Sparc specifications constitute the fundamental architectural restrictions on the Sparc CPU design. They specify instructions, data types, registers, and ways in which traps and exceptions work.

Although the architecture requires that a program generate the same results as if all its instructions were executed sequentially, the architecture provides for concurrent execution of floating-point instructions. This provision is necessary to match the 1- to 4-cycle instructions executed by the IU with the 2- to 31-cycle instructions executed by the FPU. (See Tables 1 and 2. An explanation of the modes in Table 2 appears later.)

Table 1.
Clock cycles per instruction.

Instruction	Cycles
Branch on integer/floating-point condition codes:	
Taken	1
Not taken	2
Jumps and returns	2
Load single word or smaller data sizes	2
Load double word	3
Store single word or smaller data sizes	3
Store double word	4
Launch floating-point to FPU	1
All others: shifts, logical, integer math, and so on	1

Table 2.
Clock cycles for each floating-point operation in the FPU as a function of mode.

	Throughput (cycles)			
	Mode			
Operation	00	01	10	11
Add, subtract, compare	2	2	3	3
Moves, negates, absolutes	2	2	3	3
Conversions	2	2	3	3
Single-precision multiply	2	2	3	3
Double-precision multiply	3	4	5	5
Single-precision divide	8	11	12	15
Double-precision divide	13	20	21	25
Single-precision square root	11	16	17	21
Double-precision square root	16	23	24	31

The key to the success of this matching is the use of a floating-point queue to hold pending floating-point instructions.

When the IU encounters a floating-point instruction, it instructs the FPU to enter this instruction into the queue to be executed. The IU instructions then execute as if the floating-point instruction had executed, unless a hold occurs.

Both the IU and the FPU load instructions and addresses from the system address and data/control buses (Figure 1) during an instruction fetch so they are available on the FPU for queuing. This architecture design divides the floating-point instructions into two categories—loads and stores, and operations.

Loads and stores must proceed in lockstep with the IU pipeline since the IU generates addresses. Operations wait in the queue for later execution.

Holds

The just-described architecture can result in holds (which stop IU instruction execution) under two conditions.

Condition one. The first condition occurs when a dependency exists. Such a dependency might occur when the execution of one instruction depends on the result of previously uncompleted instructions.

These holds occur in four cases:

- *When the IU executes a load to a register in the FPU file.* The IU must freeze if a floating-point operation executing, or in the queue, uses the register as source or destination. The IU waits until the operation completes.
- *When the IU executes a store of a register in the FPU file.* The IU must freeze if a floating-point operation executing, or in the queue, uses the register as a destination. It freezes until that operation is complete.
- *When a floating-point compare instruction executes.* The IU may have an instruction such as a branch that depends on the condition codes generated by the floating-point compare instruction.
- *When the IU has an instruction to load to or store in the floating-point state register that controls the rounding mode/exception enabling and holds the status.* All prior floating-point operations must complete first.

The only way the designer can minimize the hold time created by any of these four cases is to minimize (as much as possible) the latency of the floating-point operations for a given system clock frequency. Latency is the number of clock cycles from the start of the operation until the result is obtained.

Condition two. A second condition prompting a hold occurs when the queue is full and another floating-point instruction is encountered. The IU must wait until the FPU accepts the current floating-point instruction.

Designers can minimize this kind of resource hold by providing a deep queue and a floating-point pipeline. In fact, the hold will never occur if the pipeline is deep enough that an instruction completes every clock cycle and the queue has a matching depth. This practical design for addition and multiplication instructions is not as useful for the divide and square-root instructions that have high latency. The FPU queue depth is two.

Since FPU instructions occur significantly less frequently than IU instructions, the performance of the system in most applications does not significantly degrade if an FPU has a throughput even one third that of the IU. In one study on a RISC processor executing SPICE—a large double-precision numerical program—floating-point instructions made up 12 percent of the instructions executed.[2]

ALU data path

To minimize its math unit's latency, the FPU uses a highly parallel architecture. This architecture requires

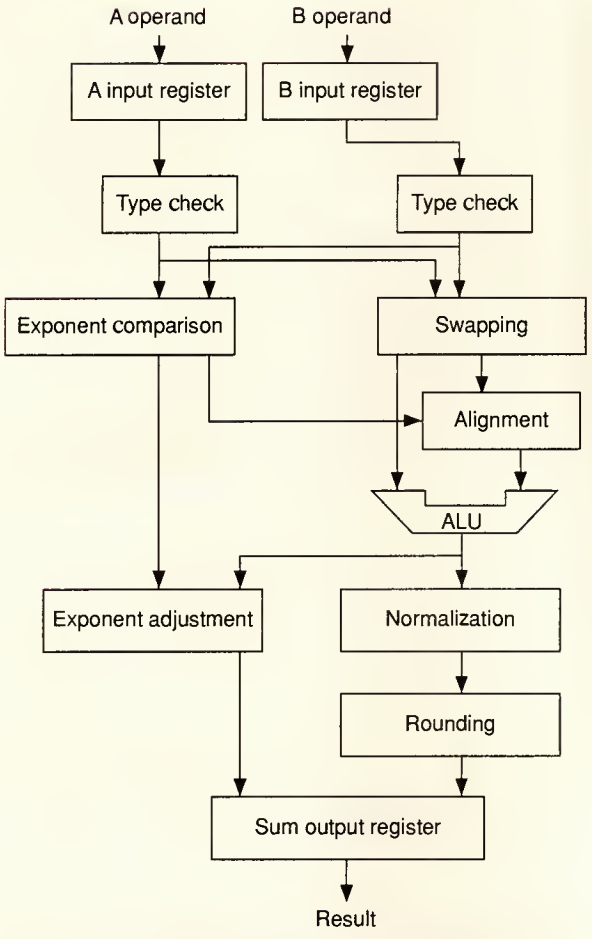


Figure 3. The ALU data path. The FPU needs separate add and multiply hardware.

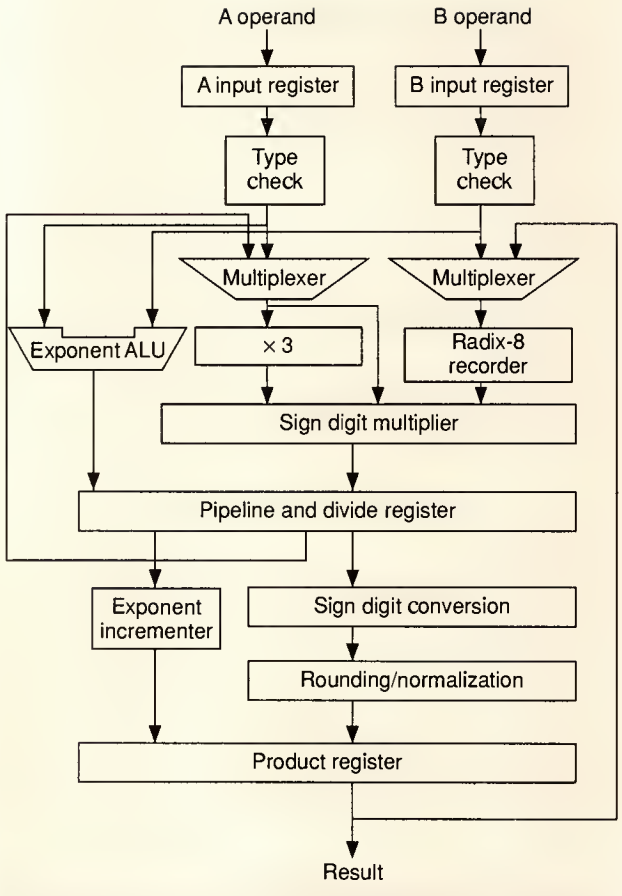


Figure 4. Data path of the multiplier unit.

separate math units to optimize additions and multiplications. An arithmetic unit performs all instructions except multiply, divide, and square root, which the multiplier unit performs. Figure 3 shows the data path of the arithmetic unit. The data path of the multiplier unit appears in Figure 4, and the box carries a description of the data flow.

As shown in Figure 3, the floating-point ALU can be divided into the functional blocks necessary to perform a floating-point addition. The data flows from top to bottom. First, both operands are reformatted and classified in the type check blocks. This classification satisfies the IEEE Std.-754 formats such as infinities, not-a-numbers, and denormals, which must be handled as special cases.

The next block compares the exponents to determine the amount of shift necessary to align the binary points for addition. This block also determines which exponent is larger. A swap block is then provided to swap the smaller of the two operands to the operand path that contains the alignment block. The alignment block performs a left shift of the amount determined by the exponent comparison block. The alignment block is a barrel shifter with some additional circuitry.

A fixed-point ALU performs the actual addition and also contains circuitry to calculate the amount of shift necessary to renormalize the result. A multibit renormalization is necessary if the leading bits are cancelled when one operand is of the same order of magnitude but the opposite sign of the other. A normalization block performs the actual right shift and the exponent adjustment block adds the shifted amount onto the exponent. Rounding by a possible increment takes place in the next block. Modifying and adding to these blocks implements other instructions. For instance, modifying the fixed-point ALU block and passing the operands and results through the other blocks unchanged implements logic operations.

We designed the FPU for systems with clock frequencies ranging from 25 to 50 MHz. To minimize math execution time for a given clock frequency, we made the number of clock cycles per operation (throughput mode) pin-programmable.

The ALU is one large combinatorial logic block without pipeline stages. Depending on the throughput mode, execution of any ALU operation takes either two

or three clock cycles. Since the ALU contains input and output registers, the FPU design determines the throughput for ALU instructions to be one half or one third of an operation per clock cycle. The throughput determines the rate at which floating-point instructions can be executed if no data-dependency holds exist.

The register file-to-register file latency accounts for two additional clock cycles. (This latency is the number of clock cycles before the results are stored in the register file and after the operands are fetched from the register file.) The read from the register file takes one clock cycle, and the write to the register file accounts for the second cycle. The latency determines how long holds last.

Multiplier data path

Unlike the ALU, the multiplier is divided into two stages with a register in between. The first stage of the multiplier contains a binary tree adder (33×60-bit sign digit) with a sign digit radix-8 recoder.[3] We originally chose the sign digit multiplier to provide high-speed multiplication in a compact layout. However, it proved to be one of the keys to implementing fast divide and square-root algorithms.

The second stage of the multiplier performs the conversion of the sign digit number to a sign magnitude number. It also performs the needed rounding and 1-bit normalization.

In the IEEE Std.-754 format the mantissa of a double-precision number is 53 bits. Since chip area considerations constrained the multiplier array to about half that size, we chose the width of the array to be 60 bits and the depth to be 33 bits. The width was set by the width of a double-precision operand plus six guard bits used by the divide algorithm (discussed later). We originally chose the 33-bit depth to permit integer formats in the predecessor floating-point unit chip. To multiply two double-precision numbers, the array requires two passes of the operands. During the first pass operand A is multiplied by the least significant half of operand B. During the second pass operand A is multiplied by the most significant half of operand B and added to the previous result.

We based the divide and square-root instructions on an iterative convergence algorithm that performs the operations as a series of multiplies, making use of the high-speed multiplier.[4] (This Goldschmidt algorithm is similar to the Newton-Raphson algorithm.) A special rounding cycle occurs at the end of a calculation; it involves an additional multiply to meet the IEEE Std.-754 requirement to round as if from infinite precision.

We modified the multiplier hardware to optimize it for this algorithm. For example, we provided additional intermediate result registers at the middle pipe and

Dataflow through the Floating-Point Unit

The complex flow of data for a calculation on a computer leads through many subsystems, including the input/output devices, disk memory, physical memory, cache memory, register files, execution units, and various buses. Here we detail the flow from cache memory through the FPU execution unit and back to cache memory. We trace the flow of a simple single-precision, floating-point add instruction to give you the feel of data movement through the machine. We try not to bog down the description with the details of exceptions, dependencies, traps, and other special cases. Table A lists the fragment of Sparc assembly code that we step through, clock tick by clock tick.

A design can be conveniently partitioned in terms of registers and what occurs between the registers. We can then describe the design behaviorally as

1) a set of register models whose contents potentially vary in each clock tick, and
2) a set of models that describe the combinational logic between the registers.

Table A.
Fragment of Sparc assembly code.

Assembly code	Explanation
Ld [%fp-16],%f10	Load FPU register 10 from the 16th entry in the procedure's stack in memory. The %fp register in the IU points to the top of the stack.
Ld [%fp-17],%f11	Load FPU register 11 from the 17th entry in the procedure's stack in memory.
Fadds %f10,%f11,%f12	Execute a single-precision add of FPU register 10 to FPU register 11 and store in FPU register 12 any IU instructions.
St %f12,[%fp-18]	Store FPU register 12 to memory stack in the 18th entry below the top.

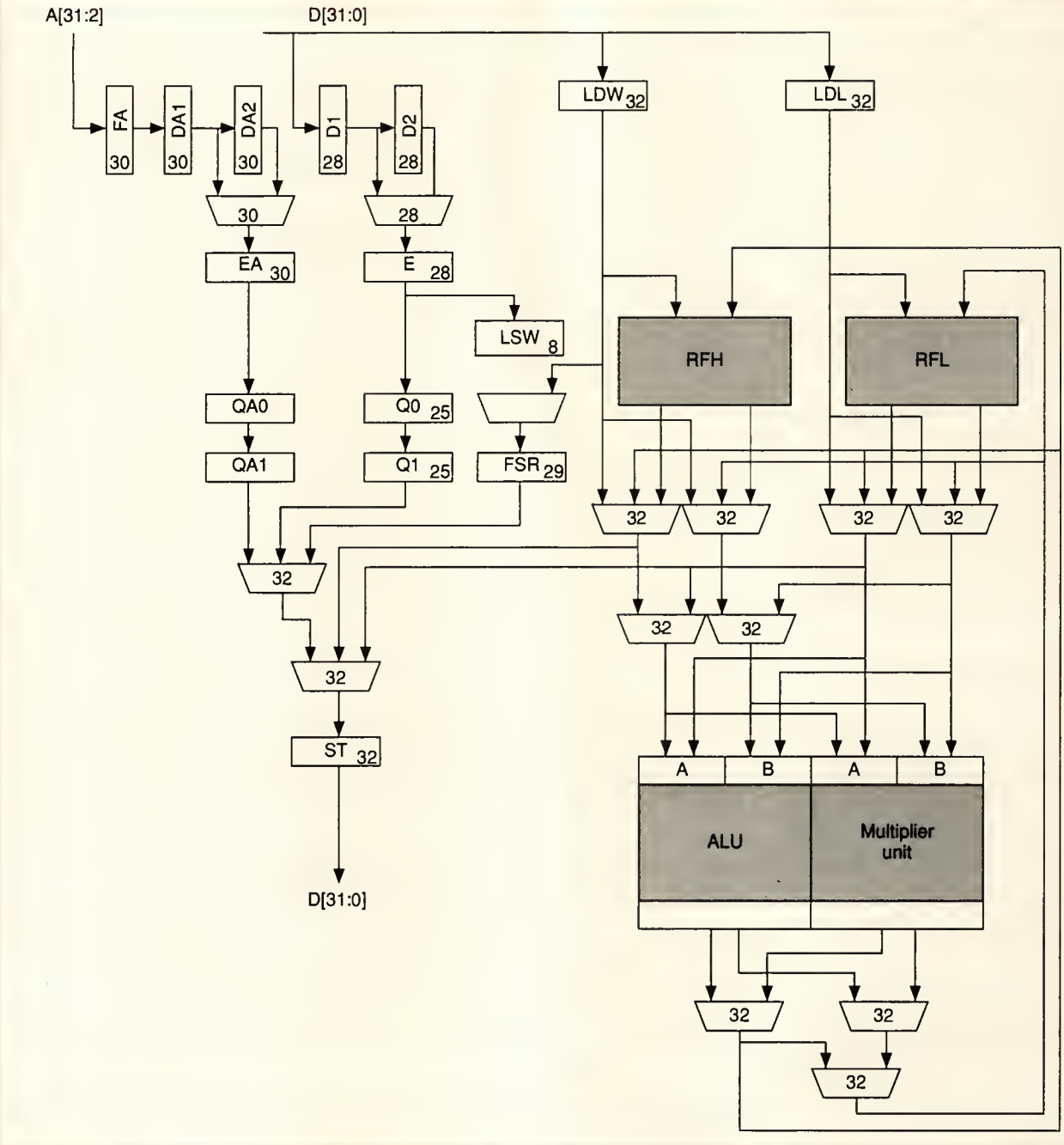


Figure A. FPU registers and major paths between them.

Several high-level computer languages for behavioral modeling such as RTL (Register Transfer Language) use this philosophy.

Figure A diagrams the FPU's registers and the major paths between them. For simplicity, we don't show the control logic and paths. The register file divides into two blocks: RFH, which contains the even registers, and RFL, which contains the odd registers. We can conveniently think of the register file as two blocks, since a double-precision word is always aligned on even boundaries with the most significant half in the even register and the least significant half in the next odd register. The blocks labeled ALU and multiplier unit are the execution units.

Figures B1 and B2 describe the contents of the registers as a function of time. When the register is en-

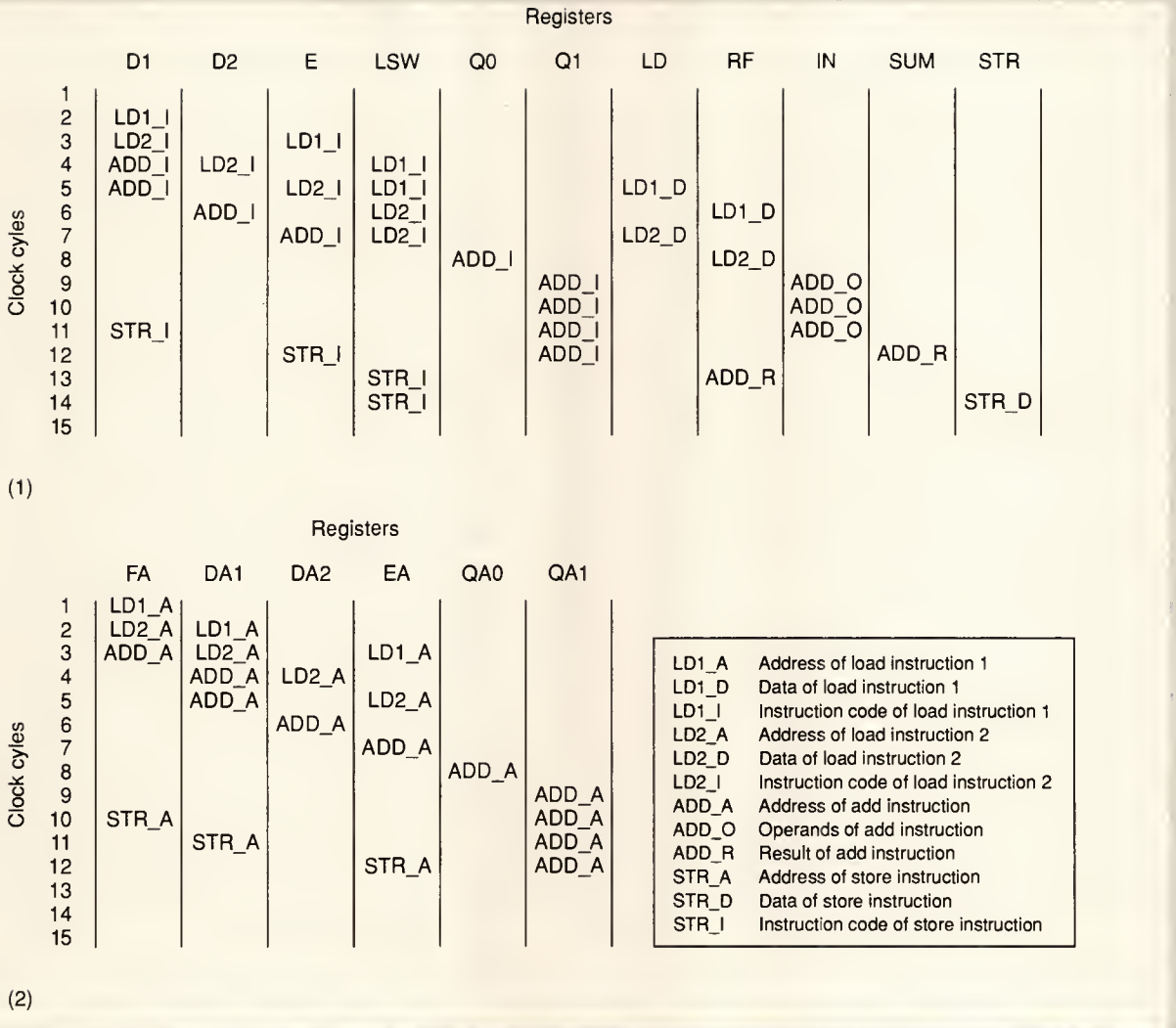


Figure B. Register contents as a function of clock cycle: Instruction and data (1) and address (2).

abled, the signal at its input is captured on the rising edge of each clock cycle. The data is then stored in that register for the entire clock period, and possibly longer if the register is not enabled at the next rising edge. Note in the figures that when a column shows text in it, that instruction or data resides in that register for that particular clock period. We assume in the timing of these figures that the cache contains all the data and instructions. Although not explicitly shown, once data is in the register file, it remains there unless overwritten by a load to that particular register of the file.

The execution of the assembly-code fragment starts with the IU sending out the address of the first load instruction LD1 on the address bus. The address is captured in register FA in the FPU at the rising edge of the first clock. This is the fetch cycle (F stage) for that particular instruction. During the clock period, the cache memory, which also captures the address, looks for the instruction in cache. If it is present, the cache outputs it onto the data bus. If the instruction is not in cache, the IU will be signaled to stall until the instruction can be supplied by the memory system. The other registers in the FPU may be changing due to previous instructions, but we don't show it here.

Since we assume LD1 resides in the cache, it latches into register D1 of the FPU on the rising edge of the second clock cycle. At the same time LD1 latches into the IU, which decodes it during clock period two. The LD1 instruction now enters the decode cycle (D2 stage) of the instruction pipeline. During this clock period, the fetch cycle of the second load instruction LD2 also executes. Note that both the address and instructions are captured and sent through a parallel path of registers forming the queue. This storing will allow exception handlers to reexecute floating-point operations that were interrupted, even though the actual program

counter may have long since passed that point in the program.

During the third clock cycle, the LD1 instruction advances to the execution (E) stage of the pipeline. In this stage the address of the data is calculated and output onto the address bus. The ADD instruction is being fetched while the LD2 instruction is captured off the data bus into the D1 register. The LD2 instruction is nominally in the D stage of the pipeline but in fact is stalled for this clock cycle because LD1 requires two cycles. The stalling of LD2 creates the extra cycle needed for LD1 to complete. LD1 needs two cycles to complete because both data and instructions use the data bus. Thus when LD1 uses the data bus for the actual data, the flow of one instruction per clock cycle is interrupted and a gap occurs in the instruction pipeline.

During the fourth clock cycle, the cache determines if the data for LD1 resides in the cache; if it does, LD1 transfers onto the data bus. This step prevents the next instruction beyond the ADD from being fetched until the following clock cycle. The LD1 instruction transfers to the load-store-write (LSW) register from the E register since it is now in the write (W) stage of the instruction pipeline. LD2 transfers to the D2 register as it has progressed to the actual decode stage after the one-cycle delay. The ADD instruction is captured in the vacated D1 register, where it will be held for two clock cycles, one for each of the previous load instructions.

During the fifth clock cycle, the data from LD1 is captured in the load data (LD) register from the data bus. The instruction stays in an extended W stage, which is its extra cycle. LD2 enters the E stage, and ADD remains stalled in the D1 stage. Another IU instruction could be in the fetch stage.

During the sixth clock cycle the data from LD1 is actually written into the register file completing the instruction. LD2 enters the W stage, and the cache unit outputs its data on the data bus. The ADD instruction enters the D2 stage for decoding.

During the seventh clock cycle, the data from LD2 is captured in LD from the data bus. The ADD instruction moves to the E stage. Here, resource and operand dependencies are checked, and the instruction pipeline could be stalled if necessary.

During the eighth clock cycle, the data from LD2 is actually written into the register file completing the instruction. The ADD instruction enters the Q0 queue register in the W stage. One operand (LDI's data) is read from the register file. The second operand (LD2's data) is being written to the file on this cycle, so a bypass around the register file to the ALU input register is activated. At the end of this clock cycle, the ADD operands enter the execution unit's input register IN. The ADD instruction exits the IU instruction pipeline at the end of this period, but it remains in the FPU queue.

During clock cycles nine to 11, the math executes in the ALU. The instruction transfers to Q1 if it is empty. At the end of clock cycle 11, the calculation completes, and the result transfers to the sum register SUM, where it is held for transfer to the register file in clock cycle 12. If the store instruction fetch occurs in the 11th clock cycle, no dependency hold results. The result will be output to the cache memory on the data bus from the STR store register during the 14th clock cycle.

added a feedback path from them to the A port of the multiplier. As a result, internally, the FPU chip can perform up to a 33×60-bit multiply and feed the results back in 30 ns.

The throughput for the multiplier is much more complex than that of the ALU because each instruction takes a different number of iterations to complete. There are two critical timing paths, one from the input register to the output register, and the other from the input register to the middle register.

The throughput mode pins can adjust the number of clock cycles allowed for each of these paths, as detailed in Table 3. Table 2 had detailed the throughput for each instruction in the various modes. In March 1990, we successfully tested the processor for functional use in a 40-MHz system in throughput mode I1 and plan 50-MHz testing for later this year.

Since divide and square-root operations have tens of throughput and latency cycles, they can easily result in resource or dependency holds. The holds become significant if the application involves a high frequency of divide and square-root operations. The latter occurs, for example, in ray-tracing graphics. The FPU offers 1.5 to 4 times higher throughput than other CMOS floating-point processors for divide and square-root operations.

Four trap types

Traps stop the execution of a program to jump to software routines for handling data-dependent errors or to execute instructions not implemented in the hardware. Without this software, the system will not fully conform to IEEE Std. 754. Four types of floating-point traps are possible: IEEE Std.-754 exceptions, unfinished floating-point traps, unimplemented floating-point traps, and sequence errors.

An IEEE Std.-754 trap occurs when an IEEE Std.-754 exception (invalid math operation, underflow, overflow, divide by zero, and inexact) occurs. The user enables the trap on that exception by setting the trap enable mask in the state register.

An unfinished floating-point operation trap occurs

when the FPU cannot complete a floating-point operation that it has started. Such situations may occur when some particular nuance of the standard was not implemented. Perhaps the performance gain from specialized hardware was not sufficient to justify the additional complexity. As an example, consider denormalized data presented to the multiplier. This type of operand does not occur frequently enough to justify the additional latency and hardware required to perform a barrel shift on the inputs of the multiplier.

The third trap type, called an unimplemented floating-point trap, occurs for the FPU only with extended-precision instructions. These instructions do not occur frequently enough to justify the wider data path or state machine needed to implement them. Also, this trap permits the floating-point instruction set to be expanded with software, as any such floating-point instruction will trap unimplemented. The set will be emulated in software by the trap handler.

Finally, the fourth type of trap, a sequence error, occurs when the FPU receives a request to execute a floating-point operation or a load instruction after an exception is signaled and acknowledged, but before it is cleared.

Three operating modes

The FPU can operate in one of three operating modes: execution, pending exception, and exception. After reset, operations begin in execution mode—the normal mode of operation.

When the FPU signals the IU that a floating-point exception trap should be taken, the FPU moves into pending-exception mode. The IU takes the trap when the next floating-point instruction executes. This mode effectively synchronizes the exception, which is issued asynchronously to the IU.

When the IU takes the trap, the FPU changes into exception mode. The FPU returns to execution mode as soon as the trap handler empties the queue via store floating-point queue instructions.

How good is the FPU?

Floating-point benchmarks measure a system's floating-point performance under restrictive conditions and depend on many system parameters. For applications dominated by high throughput and low-latency instructions (multiplies and adds), the floating-point processor often does not limit the system performance.

For example, consider the common Linpack benchmark in which the execution time is dominated by the inner loop that scales a vector **X** by a factor *A* and adds the scaled vector to another vector **Y**.

A typical compiler might unroll this operation four times. Assuming a large-enough loop that one vector is not contained in the register file, Sparc chips use 26 instructions, eight floating-point loads, four floating-point stores, eight floating-point operations, and six non-floating-point instructions in the assembly code of the inner loop. For a TMS390C601 IU, a double-precision load takes three cycles, and a double-precision store requires four cycles.

Table 3.
Maximum clock frequency for the throughput modes.

Mode	Path 1* (clocks)	Path 2** (clocks)	Maximum frequency (MHz)
00	2	1.0	25
01	2	1.5	33
10	3	1.5	37
11	3	2.0	50

* Input register to output register
** Input register to middle register (multiplier only)

Although the floating-point operations take more than one clock cycle, they count as one if there are no holds because the IU dispatches them in one clock cycle. Therefore, the loop takes 54 cycles to be executed by the IU. For a resource hold not to occur, the FPU will have to start an operation every six [int(54/8)] cycles or less. Here, we assume the compiler does a good job of scheduling the floating-point operations evenly among the other instructions.

Since the floating-point operations are adds and multiplies that can be started every two or three clock cycles, the FPU does not degrade system performance. However, if the operations being performed were divides or square roots, their worst throughputs would result in resource holds, which would degrade system performance.

In the double-precision Linpack example, the inner loop executes at 7.4 Mflops for a 50-MHz system. Because of the overhead of the rest of the code and cache misses, actual Linpack tests execute at 5.5 Mflops. Note that loads and stores are costly. For applications in which many floating-point operations can be performed per memory operation, the FPU throughput will be more important for system performance.

The Linpack benchmark does not have significant data dependencies. But, as described earlier, data dependencies can result in holds which, in turn, can degrade system performance. In any case, one 33-MHz system incorporating the TI chip-set benchmarks executed at 11.9 MWhetstones per second and 3.6 double-precision Linpack Mflops. The peak Mflops rating for

this system when executing any combination of data-independent ALU or single-precision multiply instructions is 11.1. These numbers should scale linearly with clock frequency.

A major objective of the TMS390C602A design was to optimize the system performance for given circuit and technology limits. A cooperative effort between the system vendor and the chip manufacturer allowed the chip designers to understand where the circuit realities impacted the system performance. Open communication between the designers who understood the system constraints and designers who understood the device constraints helped us to achieve our goal.

For example, we balanced the floating-point execution throughput to system data bandwidth and the instruction frequency and optimizing register-file-to-register-file latency for clock frequency. We also dramatically reduced the transistor count and die area by optimizing the interface for the specific job to be done, It also required the close cooperation of the designers. As an illustration, the die area of the 602A is 39 percent smaller than a design in which the preceding chips were simply integrated together after stripping the floating-point math chip's input and output. Further, the designers increased the clock frequency of the optimized design by 50 percent without significantly impacting system throughput or latency, as measured by clock cycles. We assume the same semiconductor technology for both designs in this increase. As a result of these modifications, the TMS390C602A—our next-generation solution for the Sparc floating-point unit—became much more powerful while dramatically cutting system cost.

References

1. *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif., 1985.
2. W. Hollingsworth et al., "The Clipper Processor: Instruction Set Architecture and Implementation," *Comm. ACM*, Vol. 32, No. 2, Feb. 1989, pp. 200-219.
3. N. Takagi et al., "High-Speed VLSI Multiplication Algorithm with a Redundant Binary Addition Tree," *IEEE Trans. Computers*, Vol. C-34, No. 9, Sept. 1985, pp. 789-796.
4. S. F. Anderson et al., "The IBM System/360 Model 91: Floating-Point Execution Unit," *IBM J.*, Vol. 11, No. 1, Jan. 1967, pp. 34-53.

Merrick Darley is a senior member of the technical staff in the Datapath VLSI Products Department in the Texas Instruments Semiconductor Group. He manages the Floating-Point Products branch. He participated in the definition and design of the IEEE-compatible floating-point processors including the ACT8837, ACT8847, TMS34082, and TMS390C602A. Previous assignments included digital CMOS design, silicon MESFET process development, and electron-beam and x-ray lithography development.

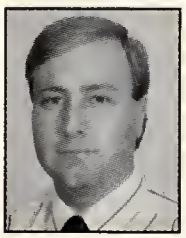
Darley holds a PhD degree from the University of Kentucky and a BA degree from Rice University. He is a member of the IEEE.

Bill Kronlage, also a senior member of the technical staff, manages the FPU CMOS Design Section in TI's Semiconductor, VLSI Datapath Products Department. He develops catalog and custom products including the Sparc floating-point coprocessors and the ACT8800 series of 32-bit CMOS processor building-block devices. Previously, he developed high-speed digital CMOS products and various Schottky TTL and I2L bipolar products.

Kronlage received a BS degree in physics from Loyola University and has completed course work toward an MS degree, also in physics, from North Texas State University.

David Bural is a member of TI's FPU CMOS Design Section. He helped design the 602A and a customer-specific floating-point processor. Previously, he designed VHSIC products using an internal gate array technology. He is interested in applying design methodologies for the rapid design of floating-point processors.

Bural holds BS degrees in electrical engineering and in computer science from Southern Methodist University.



Bob Churchill is also a member of the FPU CMOS design section. He actively participated in the design of ACT8867 and 602A floating-point processors and has worked as a VLSI product engineer. His interests include the design of the Sparc and floating-point processors.

Churchill received his BSEE degree from Michigan State University and his MSEE degree from Southern Methodist University.

David Pulling, a member of the same section, has helped design the 602A and the TMS34082 floating-point graphics processor. He has also worked as a project engineer for a digital missile guidance system. His technical interests include parallel computer architectures.

Pulling holds a BS degree in electrical engineering from West Virginia University and an MBA from the University of Dallas.

Paul Wang, another member of this section, has contributed to the design of five CMOS projects. He is interested in the design of floating-point ALUs. Wang obtained his BSCE degree from the University of Illinois at Urbana-Champaign.

Rick Iwamoto has been a member of the technical staff at Sun Microsystems, Inc., since 1986. Currently, he works on the design of next-generation Sparc-based processors. His technical interests include computer architecture, VLSI CAD tools, floating-point arithmetic, and graphics and window-based programming.

Iwamoto received the BSEE and computer science degrees from the University of California at Berkeley.

Larry Yang, also a member of Sun's technical staff, works on the design of a Sparc-based multiprocessing system. In the past, he helped design the floating-point processors of the first two implementa-tions of Sparc microproces-sors.

Yang received the MSEE and BSEE/biological sciences degrees from Stanford University. He is a member of the IEEE.

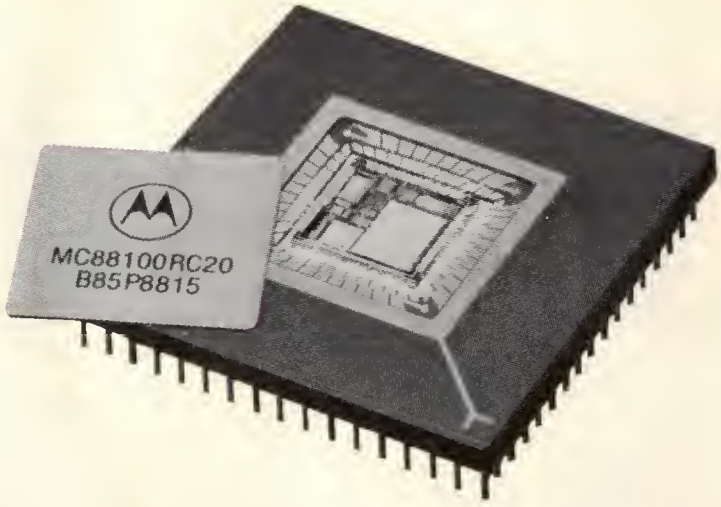
Questions concerning this article can be addressed to Merrick Darley, Texas Instruments, 8330 LBJ Freeway, PO Box 655303, Dallas, TX 75265-5303.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 159 **Medium** 160 **High** 161

FEATURE



Motorola's 88000 Family Architecture

Processors designed to support maximum instruction-set functionality contain features that impede the construction of high-performance implementations and reduce the effectiveness of optimizing compilers. A new generation of architectures emphasizes performance with pipelined data paths, cache memories, and optimizing compilers.

Mitch Alsup

Motorola

Motorola addressed the need for high-performance 32-bit microprocessors by introducing a new computer family. The initial members of the 88000 family are the 88100 processor and the 88200 cache and memory management unit, or CMMU. The processor manipulates integer and floating-point data and initiates instruction and data memory transactions. The CMMU minimizes the latency of main memory requests by maintaining a cache for data transactions and a cache for memory management translations.

A typical system consists of one processor and two identical cache chips, one servicing instruction fetch requests, the other servicing data read and write requests. We know this kind of two-memory system as a Harvard architecture.

Figure 1 shows the functional units within the 88100 processor, the memory and translation caches within the 88200, and the connections between these chips and the memory system. Many systems will use one 88200 for the code port and another 88200 for the data port.

We concentrated on maximizing performance in the chips on three levels:

- *hardware performance* by means of clock rate, pipelining, and concurrency;
- *software performance* by providing the most useful operations for high-level languages and a register-rich environment for data values; and,
- *system performance* by providing large coherent code and data caches with built-in memory management units.

Here, I explain the decision process behind the 88000 family in three interacting parts: processor, cache, and software.

The design process

In 1984 Motorola successfully demonstrated its manufacturing capability of large CMOS (complementary metal-oxide semiconductor) microproces-

0272-1732/90/0600-0048$01.00 ©1990 IEEE

sors with the 68020. This processor measured approximately 360 mils on a side and operated at a clock rate of 16 MHz. It can deliver 2.3 million instructions per second at 16 MHz with a good memory system. Although its performance was good, the 68020 averaged seven cycles for each instruction it executed.[1] Our target for the 88000 became a microprocessor capable of a sustained execution rate of one cycle per instruction.

Performance considerations. Performance is a function of the processor cycle time, number of instructions executed, and average number of cycles each instruction takes to execute.[2] The clock rate depends on the microarchitecture of the processor and the implementation technology. The number of instructions executed depends on the strength of the instruction set and the number of registers available, the capability of an optimizing compiler, and of course the algorithm being implemented. The number of cycles per instruction is a function of the microarchitecture, code scheduling, cache hit rates, and main memory effects. Therefore

CPU performance =
clock rate/(instructions executed * cycles/instruction)

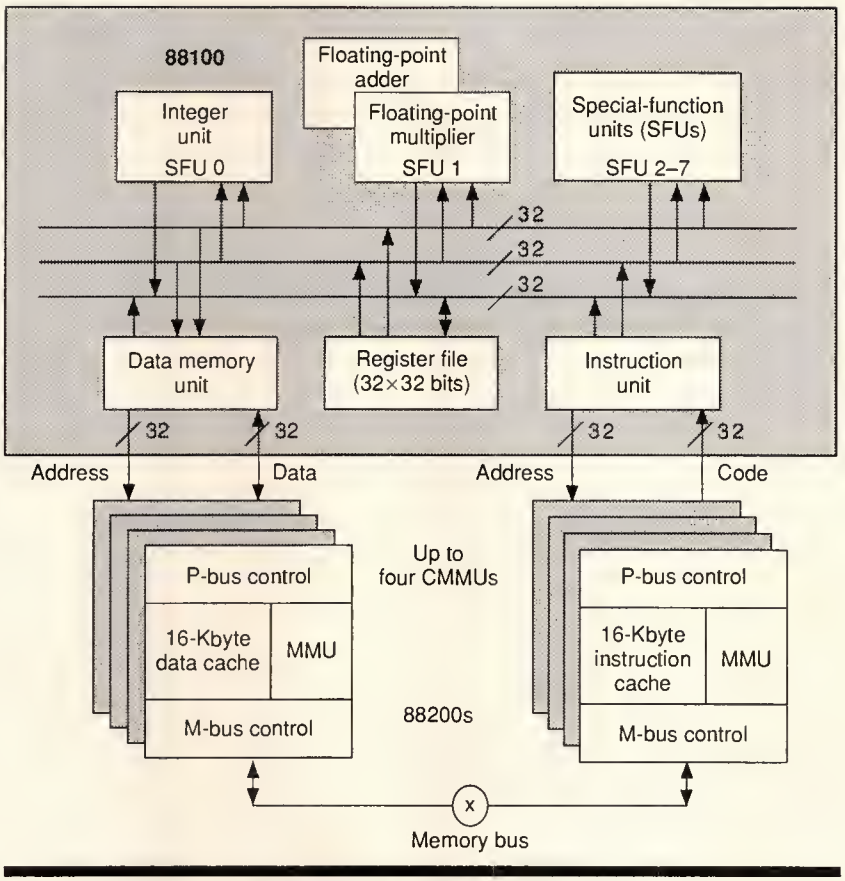


Figure 1. Block diagram of an 88100/88200 system.

Instruction composition. An instruction set is the blending of the operations applied to the operands that are producing results. It is the template on which compilers compose instruction streams and from which processors execute programs. Analysis of instruction sizes in the VAX architecture on integer benchmarks indicated an average instruction size between 29.7[3] and 30.8 bits.[4] The 68000 family shows an average instruction size of 28 bits.[5]

Both the VAX and the 68000 have densely encoded instruction sets and complex addressing modes. An operation code specifies the operation to be performed, and the addressing mode specifies the location of the operands (in a register or in addressable memory). In addition, the addressing mode can also specify side effects in the address computation (postincrementing the base register, for example). The addressing mode was, in effect, an instruction embedded in an instruction. The addressing modes were themselves encoded, register and simple memory references taking fewer bits than more exotic addressing modes. The variable size of an addressing mode made the process of fetching and decoding instructions quite complex.

Analysis of high-level-language programs indicated that most operations are simple. As a result, we chose to simplify the process of fetching and decoding instructions by making each instruction 32 bits in size. We based the instruction set on a simple paradigm—each instruction does exactly one thing. We felt that embedding addressing modes in an instruction violated this paradigm and chose to reference memory exclusively through load and store instructions.

Fast access to the operands is crucial for high-performance computing.[6] To ensure fast access in the 88100, we included a 32-entry, 32-bit register file within the processor to hold variables and other data. Since the processor can process an instruction in one cycle, and a moderate fraction of these instructions (around 35 percent) will reference memory, the 88100 dedicates a memory port entirely to instruction fetch processing. A second memory port supports data memory reference operations. This kind of two-bus system is key to supplying adequate memory bandwidth to both the instruction and the data streams.[7]

Optimizing compiler considerations. Since the 88000 architecture does not contain addressing modes,

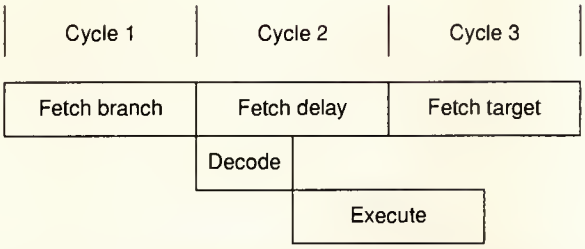


Figure 2. The instruction fetch pipeline feeding the execution pipelines.

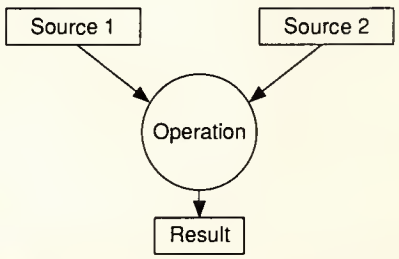


Figure 3. Triadic instruction encoding.

we felt 16 general-purpose registers, as in the VAX architecture, would be insufficient and 64 registers required too many bits to encode. The 88000 architecture provides 31 general-purpose registers, each 32 bits in size. Register 0 is hardwired to zero and therefore is not general purpose. Several of these registers have been assigned permanent duties (by software convention) as the stack pointer (r31) and the frame pointer (r30).

Register-rich architectures can pass arguments and receive results from subroutines in their registers. Subroutines that do not call other subroutines can (often) perform all of their computations in the registers. Both of these techniques speed the access to operands by avoiding the use of memory. Almost all subroutines have fewer than eight arguments and return zero or one result.[8] The 88000 architecture exploits the register set by defining a subroutine linkage convention, which efficiently uses the registers.

Pipelined microprocessors create additional opportunities for optimization.[7] Optimizers can schedule code sequences to allow the hardware to run more efficiently. The pipelined instruction fetch process requires that the next instruction address be sent to the cache as the cache sends the current instruction back to the processor. A branch instruction can influence only the subsequent cache access; the instruction in the shadow of the branch is known as the delay instruction. The data memory and the floating-point units are similarly pipelined. Note that loaded values cannot be used in the instruction immediately following the memory reference.

In Figure 2 we can see the instruction fetch pipeline feeding the execution pipelines. The branch instruction arriving at the end of cycle 1 is too late to affect the instruction fetch in cycle 2 (the delay slot). But the branch is ready to fetch the target of the branch (or the next sequential instruction) in cycle 3.

The instruction set encodes the operations on explicitly named operands and results. Figure 3 shows the triadic instruction encoding in dataflow form, a superset of the monadic and dyadic forms. It sacrifices instruction set density for nondestructive data manipulation operation capability.

A monadic instruction set encoding (as in the PDP 8) performs all computations on the single register, impacting both allocation and scheduling optimizations. A dyadic instruction encoding (as in the 68020[1]) destroys one of the operands during the computation. This operation can require move instructions and reduce the ability to schedule the code sequences against a pipeline. In a triadic instruction set, each computation can place its result into the most appropriate result register. This separation provides the compiler freedom to allocate variables to registers and to schedule code sequences efficiently.

Most instructions compute a result by applying an operation to operands. A load instruction, like any other instruction, delivers a value, but a store instruction also delivers its value from the register file to memory. Since the store value is not needed until the store address has been generated, store instructions in the 88100 read the register file during the period when one-cycle instructions would be writing their results.

An instruction that uses the result produced by an earlier instruction depends on that previous instruction. An instruction scheduler attempts to place independent instructions between dependent ones. It also avoids pipeline stalls when control is transferred by placing an instruction in the shadow of a branch instruction. A code scheduler for the 88100 averages a 12-percent performance gain over unscheduled code.

As an example of instruction scheduling, consider this description of the 88100 assembler notation. A label may be applied to an instruction by following a name with a colon (:). The instruction mnemonic follows the label, and a lowercase *r* followed by the register number designates a register. By convention, the destination register is the first register designated, followed by the source one operand register and finally by the source two operand. Therefore, the generic instruction template is:

```
[label:]        opcode  rd,rs1,s2
```

Local labels generated by several compilers precede the label name with an @ sign. To ease assembly

language programming, we access additional mnemonics for encoding conditional branch specifications. For branch-on-condition instructions, we use mnemonics for comparisons to zero, such as greater than or equal to zero (ge0). And for branch-on-bit instructions, we use mnemonics for normal comparison results, such as unsigned less than or equal to (ls).

In Table 1 we can see the effects of scheduling a sequence of instructions. Each Add instruction in the unscheduled sequence must wait for its operand to arrive from the data cache, while no instruction is delayed on the scheduled side.

Table 1.
Effects of scheduling a sequence of instructions.

Unscheduled		Scheduled	
Opcode	Operands	Opcode	Operands
Ld	r7,r31,32	Ld	r7,r31,32
Add	r7,r7,1	Cmp	r10,r6,r2
St	r7,r31,32	Ld	r8,r31,40
Ld	r8,r31,40	Add	r7,r7,1
Add	r8,r8,4	Add	r8,r8,4
St	r8,r31,40	St	r7,r31,32
Cmp	r10,r6,r2	Bb1.n	lt,r10,@L2
Bb1	lt,r10,@L2	St	r8,r31,40
Total cycles: 13		Total cycles: 9	

Processor considerations. The 88100 has a short cycle time, barely longer than the speed of an integer adder or a cache memory access. Memory references and floating-point operations take significantly longer than this short cycle time. While these facts precluded our making all instructions finish execution in one machine cycle, they do not preclude the processor from initiating an instruction each cycle. The 88100 resorts to concurrency and pipelining to achieve its performance goals.

Function units. The CDC 6600 introduced the concept of multiple concurrent, independent function units connected to a multiported register file under control of a central scoreboard.[9,10] CDC augmented this concept in its 7600 with the inclusion of pipelining in the function units.[11]

The 7600 dispatches operations that take longer than one machine cycle to a function unit, which can process the operation concurrently with other operations. The function units are further sliced into machine cycle length segments. An instruction flows through this pipeline and delivers its result to the register file with many instructions executing simultaneously. The register file monitors this concurrency and prevents data-dependent instructions from issuing. Together these mechanisms allow instructions to be issued each machine cycle and complete at a later point in time to avoid needless stalls.

These concepts formed the basis for the data path architecture of the 88100. We designed five function units in the 88100: integer, data memory, floating-point addition, multiplication, and instruction. Each unit operates independently under its own control, and the register file acts as the central repository of data values being manipulated.

The scoreboard in the CDC 6600 allows several concurrent memory operations. This machine, with a memory cycle time of 10 processor cycles, can complete eight memory requests in one memory cycle. The multiported memory system requires many address and data buses between the memory and the processor—many more than we could afford. Instead of this kind of high-bandwidth memory system, the 88000 utilizes a low-latency memory system.

Cache memory. The IBM 360/85 introduced the concept of a cache memory system:[12] a small, fast memory that can process a memory request in one cycle if the data resides in the cache. When the data is not in the cache, the processor must access main memory and move the data to the cache for fast accesses on subsequent requests. Cache memories reduce access times for memory requests because memory reference patterns have spatial and temporal locality.[13] Spatial locality indicates that when a location is referenced, other data near this reference may also be referenced. Temporal locality indicates that once an item is referenced, a good probability exists that it will be referenced again in the near future. The 88200 cache and memory management chip provides the 88100 processor with single-cycle access to code and data requests.

A cache memory system has an average access time $\underline{Tac}$ based on the access time of the cache Tac_{cache} and of main memory Tac_{memory}. Hits are completed after a cache access delay, while misses are completed after a main memory access delay. The cache may not be available for access when an outside party updates its contents, and access to main memory may be delayed by bus arbitration.

The following equation shows the relationship between the average access time seen by the processor and the access times demonstrated by the caches and memory systems.

$$\underline{Tac} = \text{Hit} * \underline{Tac}_{cache} + \text{Miss} * \underline{Tac}_{memory}$$
$$\underline{Tac}_{cache} = Tac_{cache} + \text{Snoop}$$
$$\underline{Tac}_{memory} = Tac_{memory} + \text{Bus}$$

Here, Snoop is the fraction of time that a cache may be busy for reasons other than processor requests (can be zero), and Bus is the fraction of time that either the bus or the memory is unavailable due to contention, re-

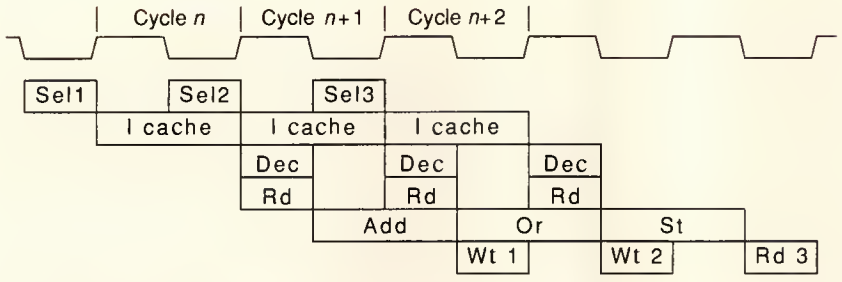


Figure 4. Master pipeline for the 88100 processor.

freshing, and dynamic RAM precharging.

In the interest of keeping the instruction set simple, the 88000 architecture provides no direct support for input and output devices. We mapped I/O devices into the normal memory address space; load and store instructions access the devices. The 88200 MMU can inhibit caching of these locations.

Master pipeline. A major step in the architecture design was the design of the master pipeline around the register file. This file contains two ports dedicated to sourcing operands and a third port used for writing results into the register file or sourcing data to be stored into memory. Since instruction decoding and register file access is allotted one half of a machine cycle, instruction decoding must be simple, and the register fields in the instruction must be in a fixed position. Instruction and data cache accesses take a full cycle. Input operand selection, computation, and the result's bus drive take another cycle and allow back-to-back data-dependent operations. The processor writes the result into its register concurrently with the beginning of the next instruction.

Figure 4 illustrates three instructions flowing down this master pipeline from the time of instruction address selection to the writing of the result into its register. A cycle begins with the transmittal of the selected instruction address to the instruction cache. The Sel component selects the address to be fetched from the instruction cache, or I cache. Instruction decoding (Dec), register read (Rd), and result forwarding (not shown in the figure) take place simultaneously, followed by instruction execution (Add, Or, St) and register write (Wt). Note that the store instruction St reads the register file during the normal write slot interval.

The pipelined instruction fetch process lets the processor decode one instruction while the access of the following instruction is already in progress. The branch instruction can replace the subsequent cache access with the branch target address if the branch is taken. So that this access is not wasted, the 88000 architecture permits optional execution of the instruction following the branch. If the compiler can find a useful instruction to place after the branch, it sets the execute bit in the branch instruction, and the delay slot executes before control arrives at the target location. If the branch instruction doesn't transfer control, the delay slot instruction always executes. The assembler recognizes this delayed form by following the branch mnemonic with *.n* (bb1.*n* eq,r10,@L30).

Data-driven operation. We recognized that the 88000 should be designed so that each implementation could freely exploit the implementation technology available. Each implementation designs function units with latencies as short as practical; therefore, the latency of any particular unit should vary over a small range. Since software cannot always detect the exact latencies for the function units, the hardware itself must assume responsibility for the correct sequencing of the instruction stream. Scheduling software can move dependent operations as far apart as possible, filling the spaces with other nondependent instructions, and achieve near optimal schedules across several implementations.

Register interlocking. Since all computed values in the 88000 are delivered to a register, the processor can monitor an instruction's completion time by the arrival of its result. This capability allows the processor to continue issuing instructions, even though earlier instructions have not yet completed.

The register file maintains a presence bit for each value associated with a register. Any instruction that delivers a result causes the presence bit of the destination register to be set. When the value is delivered to the result register, the bit clears. If an instruction requires an operand that is not present, the 88100 prevents the instruction from issuing. In the absence of operand conflicts, the 88100 can have up to 11 instructions in process at one time.

Figure 5 shows how an instruction is wired to the register file during the register read portion of the master pipeline. Note that the forwarding multiplexers allow a result to be used immediately as an operand. The state accessed by the register access determines whether the instruction proceeds to execute or waits (because it is data dependent).

Instruction-set composition. Many of the operands found in high-level-language programs (and their support environments) are small constants.[6] These operands carry out initialization and incrementation assignments. They also act as offsets from a structure pointer to a field within the structure and as offsets to a branch target. Although small binary constants are the norm,

moderately large constants occur sufficiently often to warrant inclusion. The 88000 architecture provides 16-bit immediates for all of the integer arithmetic, memory addressing, conditional branch offsets, and logical instructions.

A quick language survey found more than 32 instructions requiring immediates. Therefore, the 88000 architecture allows 64 major operation encodings from a 6-bit field. Two 5-bit register fields (result and first operand) follow this opcode field, leaving 16 bits for the immediate or second register field. When the instruction does not specify an immediate field, the architecture borrows these 16 bits and uses 11 bits for the minor operation code field and the remaining 5 bits as the second operand register.

Figure 6 shows how we chose to lay out the bits in the instruction encoding. The major opcode field uses 6 bits and controls five decoding formats. All three register instructions use the triadic form. We encoded the bit field instructions (Bit), which use only 10 bits of immediate data, separately from the general-purpose immediate instructions (Immed) with 16 bits of immediate data. Conditional branches (Cbr) borrow the destination register field to encode the condition on which to transfer control. Direct branches (Br) and branches to subroutines (Bsr) provide a 26-bit (instruction pointer relative) offset to offer 128-Mbyte program sizes without special linkage editing.

Control flow. Many earlier architectures use the concept of a condition code register, which is modified as a side effect of most instructions. Conditional branch instructions examine the condition code to effect branch decisions. When condition codes are set from nearly every data processing instruction (for example, IBM 370,[14] DEC VAX[15]), they form an implicit data dependency between data processing instructions and branch instructions. Thus, they reduce the compiler's ability to schedule instructions. Since the 88100 does not always complete instructions in the order that they are issued, managing a condition code could have added both complexity and hindered performance. We found two solutions, control the setting of the condition code[16] or do not implement a condition code model.[1]

We chose to implement a noncondition code model. Our branch instructions examine data held in a general-purpose register and base their decisions on a code from the branch instruction and the value in the register. Many high-level-language comparisons, especially in the C language, are made to zero. In C a pointer with a value of zero cannot be dereferenced, and a null character terminates character strings. Therefore, we created a conditional branch instruction that compares its operand to zero and can branch if the operand is equal to zero, less than zero, or greater than zero (in both signed and unsigned forms).

Two additional branch instructions transfer control by examining a single bit in an operand. One transfers

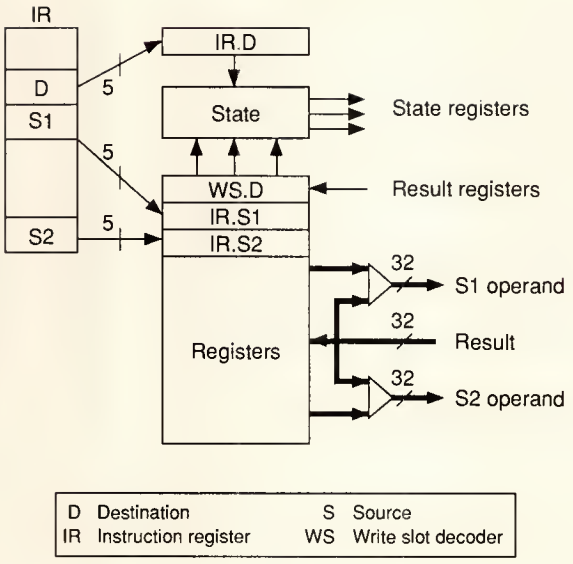


Figure 5. Wiring an instruction to the register file.

6 bits	5 bits	5 bits	11 bits		5 bits
Triadic	D	S1	Op code		S2
Bit	D	S1	Op	Width	Offset
Immed	D	S1	Immediate		
Cbr	M/B	S1	Offset		
Br/Bsr	Offset				

M/B Mask or bit

Figure 6. Bit layout in the instruction encoding.

control if the bit is clear, the other if it is set. These instructions are useful in device drivers and service processes that check rights and options as they provide service. They are also useful in high-level languages when the tested variable masks a number whose binary value contains a single bit (if(flag & 1)).

Boolean predicates. Most high-level languages require a means to compare integer and floating-point values against each other and provide a variety of control flow decisions based on the outcome. Since the 88000 does not have a condition code register, we decided that the comparison instruction (now patented) should deliver its result to a general-purpose register. A register contains 32 bits and allows many useful comparison results to be held in a bit vector of Boolean predicates. The integer compare instruction returns 10

bits of a result, including both signed and unsigned Boolean values. The floating-point compare instruction returns 12 bits of a result as IEEE Std.-754[17] floating-point operands may not be comparable (not a number).

A branch-on-bit instruction can examine one bit and decide to transfer control. Or a bit-manipulation instruction can extract the predicate and express it as an unsigned number {0,1} or as a signed number {– 1,0} depending upon the high-level language's preference. Thus the comparison instructions provide means for controlling the flow of the program and for generating individual truth values. Since the Boolean vector is delivered to a general-purpose register, the compiler can treat the vector as data, making it suitable for the same optimizations as any other data.

Figure 7 demonstrates an instruction comparing two operands and creating a 10-bit result that represents all 10 ways of relating two (integer) operands with each other. Primarily branch-on-bit instructions use this result to transfer control.

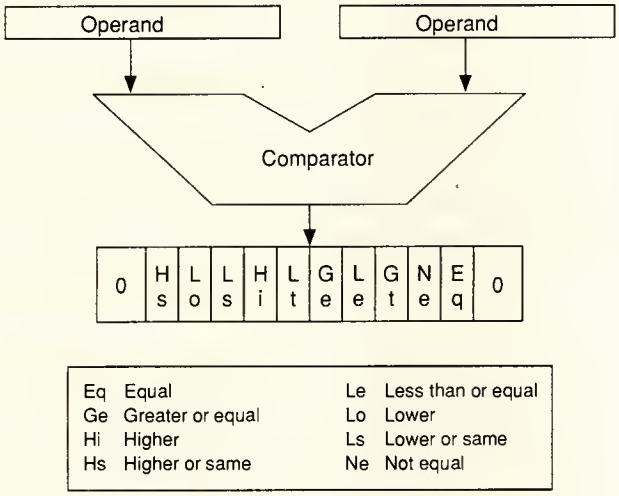


Figure 7. Comparison instruction.

Integer instructions

Tables 2 through 7 list the instruction set of the 88100 architecture.

Integer arithmetic, the backbone of any instruction set, also provides for register data movement and address computations, or it can be used to synthesize missing functionality. The 88000 architecture processes character data by first loading 8-bit memory locations into a 32-bit register and then storing 32-bit register data into 8-bit memory locations. While in a register, the processor treats character data as integer data.

Signed integer arithmetic implements with 32-bit two's-complement numbers providing a binary range of –2147483648 to +214783647. Results outside of this range cause overflow traps. Unsigned integer arithmetic exists in the range of 0 to 4294967295 but does not cause overflow traps. To support number systems larger than 32 bits, we provide multiprecision versions of addition and subtraction instructions.

All of the operations in Table 2 except multiplication and division complete in one cycle. Integer multiplication takes only four cycles, while integer division takes 38 cycles.

The 88100 implements a large variety of simple logical operations. The immediate versions of the instructions can apply the 16-bit immediate (imm16) to the upper or lower 16 bits of the operation, with the other 16 bits being treated as zero. Since the logical And instruction clears all of the bits not covered by immediate to zero (thus losing information), we included a variant (mask) that does not destroy the unprocessed bits. Triadic logical instructions allow optional bit inversion of the second operand. This method allows two logical operations to be processed in one instruction, the bit inversion, and the specified operation. Bit clearing is one example of this useful extension. All of the logical data-manipulation instructions in Table 3 complete in one cycle.

Absolute memory references can be constructed in two instructions; the first

Table 2.
Integer data-manipulation instructions.

Opcode	Operands	Function
Add	rd,rs1,imm16	Signed addition
Addu	rd,rs1,rs2	Unsigned addition
Add.c[io]	rd,rs1,rs2	Multiprecision addition
Addu.c[io]		
Cmp	rd,rs1,imm16	Comparison
	rd,rs1,rs2	
Div	rd,rs1,imm16	Signed division
Divu	rd,rs1,rs2	Unsigned division
Mul	rd,rs1,imm16	Multiplication
	rd,rs1,rs2	
Sub	rd,rs1,imm16	Signed subtraction
Subu	rd,rs1,rs2	Unsigned subtraction
Sub.c[io]	rd,rs1,rs2	Multiprecision subtraction
Subu.c[io]		

instruction is a logical Or of zero (r0), and the upper 16 bits form the immediate field. The remainder of the address forms in the load or store instruction as needed.

The bit-manipulation capability in the 88100 capitalizes on the idea that shift instructions could be subsets of a more capable bit manipulation. Shift right (left) instructions use an extract (make) instruction and specify a width of 32 bits (encoded as zero). This capability quite handily extracts exponents from floating-point data items and Boolean data from the compare instruction. All bit-manipulation instructions (listed in Table 4) execute in one cycle. We also use the extract instruction (ext, extu) for right-shifting and the make instruction for left-shifting operations.

Data transfer instructions. Data transfer instructions handle all transfers to and from the register set. Memory reference instructions move data residing outside the processor. High-level languages require efficient access to several kinds of memory structures. Local variables usually move to the stack and are referenced with immediate offsets from the stack or frame pointer. Scalar global data is referenced with immediate offsets from a pointer initialized by the linker. Fields within aggregated data structures are referenced by offsets from a structure pointer. Arrays of scalar data can be indexed by adding an array pointer to a suitably scaled index value.

The 88000 architecture provides a full complement of load and store instructions. It supports signed and unsigned (zero extended) bytes (8 bits), signed and unsigned half words (16 bits), full words (32 bits), and double words (64 bits). A double-word load instruction takes one issue cycle and two result cycles.

All memory reference instructions form their address by adding a base register to a second operand. Integer arithmetic will already have provided for the addition of either a 16-bit immediate or a second 32-bit register as the second operand. We felt memory reference instructions should do likewise, and provide both immediate offsets and register indexing. We included scaled indexing to support the manipulation of arrays.[4] Since the address adder only performs addition, we decided to incorporate a scaling multiplexer into its data path. The load address instructions deliver the computed address as an integer result.

The exchange with memory instructions provides atomic access and modification of memory. Viewed from the memory system, these instructions appear to be a load followed by a store with no intervening accesses allowed. Viewed from the register file, the instructions appear to be a store followed by a load. Thus the register and memory data are exchanged. Since the value written to memory resides in a register, forms of "test and set" and interlocked, linked list-manipulation primitives can be implemented.

The immediate form of the memory reference instructions listed in Table 5 on the next page provides a 16-bit immediate value that is added to the base pointer to compute the referenced address. The first three register forms add the two specified registers together to constitute the reference address. Meanwhile the second form scales the second operand (index) by the size of the data before adding it to the base register. An operating system may use the user space reference option [.usr] to reference data from the user's address space.

Table 3.
Logical data-manipulation instructions.

Opcode	Operands	Function
And[.u]	rd,rs1,imm	Destructive And
Mask[.u]		Nondestructive And
Or[.u]		Inclusive Or
Xor[.u]		Exclusive Or
And[.c]	rd,rs1,rs2	And
Or[.c]		Inclusive Or
Xor[.c]		Exclusive Or

Table 4.
Bit field-manipulation instructions.

Opcode	Operands	Function
Clr	rd,rs1,imm10 rd,rs1,rs2	Clear bf* to zeros
Ext	rd,rs1,imm10 rd,rs1,rs2	Extract signed bf
Extu	rd,rs1,imm10 rd,rs1,rs2	Extract unsigned bf
Ff0	rd,rs2	Find first zero bit
Ff1	rd,rs2	Find first one bit
Mak	rd,rs1,imm10 rd,rs1,rs2	Make a bf
Rot	rd,rs1,imm10 rd,rs1,rs2	Rotate a bf
Set	rd,rs1,imm10 rd,rs1,rs2	Set a bf to ones

*Bit field

Control transfer instructions. The 88000 architecture provides a full complement of control transfer instructions. Unconditional branch instructions use the condition field and the operand specifier field as a 10-

Table 5.
Memory reference instructions.

Opcode	Operands	Function
Ld.b	rd,rs1,imm16	Load signed 8 bits
Ld.ub		Load unsigned 8 bits
Ld.h		Load signed 16 bits
Ld.uh		Load unsigned 16 bits
Ld		Load 32 bits
Ld.d		Load 64 bits
St.b		Store 8 bits
St.h		Store 16 bits
St		Store 32 bits
St.d		Store 64 bits
Ld.b[.usr]	rd,rs1,rs2	Load signed 8 bits
Ld.ub[.usr]	rd,rs1[rs2]	Load unsigned 8 bits
Ld.h[.usr]		Load signed 16 bits
Ld.uh[.usr]		Load unsigned 16 bits
Ld[.usr]		Load 32 bits
Ld.d[.usr]		Load 64 bits
St.b[.usr]		Store 8 bits
St.h[.usr]		Store 16 bits
St[.usr]		Store 32 bits
St.d[.usr]		Store 64 bits
Xmem.b		Exchange 8 bits
Xmem		Exchange 32 bits
Ldcr	rd,crs	Load control register
Stcr	rs1,crs	Store control register
Xcr	rd,rs,crs	Exchange control register

Table 6.
Transfer control instructions.

Opcode	Operands	Function
Jmp[.n]	rs1	Jump to address
Jsr[.n]		Jump to subroutine
Bb0[.n]	bit,rs1,label	Branch if bit is zero
Bb1[.n]		Branch if bit is one
Bcnd[.n]	cond,rs1,label	Branch on condition
Br[.n]	label	Branch
Bsr[.n]		Branch to subroutine
Tb0	bit,rs1,vector	Trap if bit is zero
Tb1		Trap if bit is one
Tbnd	rs1,imm16	Trap out of bounds
	rs1,rs2	
Tcnd	cond,rs1,vector	Trap on condition
Rte		Return from exception

bit extension to the normal branch offset, allowing direct branches over 128 megabytes in displacement. Jump instructions transfer control to an address specified in an operand register. Branch and jump-to-subroutine instructions transfer control to their specified target addresses while delivering a return address to a register (r1). Traps into the operating system instruction request system services, and a privileged instruction returns control from the operating system to another application.

Table 6 lists the transfer-of-control instructions. They include direct jumps, relative conditional and unconditional branches, conditional traps to the operating system, and means for the operating system to return control to user programs.

Floating-point instructions. The 88000 architecture adopts IEEE Std.-754 floating-point arithmetic. Floating-point processing divides into two processing units, the floating-point adder and the floating-point multiplier. The floating-point adder is pipelined in both single- and double-precision operations. The multiplications flow through the pipeline at a single-cycle rate in integer and single-precision, a two-cycle rate in mixed mode, and a four-cycle rate in double-precision operations. The 88100 also supports conversions to and from integer.

The IEEE floating-point data-manipulation instructions listed in Table 7 process single-precision and double-precision operands and produce single- and double-precision results. The size (siz) specification denotes each operand's size independently. For example, fadd.sds is an instruction that produces a single-precision result from a double-precision operand and a single-precision operand.

The processor

The 88100 processor emulates the 88000 architecture by partitioning the operations among function units and controlling access to the register file. The master pipeline sequences instructions through the instruction unit, the register file, and the decoder, and into the data manipulation units. Forwarding logic detects that the register number of the current result is one of the current operands; it forwards the result to the function unit, thereby avoiding the delay of the register file.

Figure 8 shows the time and placement of the duties within the master pipeline. Instruction processing begins with the selection of an

instruction address to fetch from the cache and ends (typically) with a result being written into the register file. The arriving instruction is decoded, the register file read, and any branch target address computed. The operation proceeds on the operands, with the result being written into the register file.

The pipelined data memory unit can start a new memory reference each clock cycle, subject to the performance of the cache memory. This pipeline takes three cycles: address computation (one cycle); address transfer to the cache, or drive (one-half cycle); cache access (one cycle), and data transfer between the processor and the cache, or receive (one-half cycle). In Figure 9 on the next page we can see the way the memory unit pipeline fits onto the master pipeline. The instruction is decoded and register file operands read (as before). The address moves to the data cache, and the returned data passes through a multiplexer to extract bytes and/or half words as required.

We decided not to incorporate the data memory unit into the master pipeline so that the processor could continue to issue instructions when the access missed the cache. This capability is useful when the compiler is smart enough to touch memory locations a loop ahead of the computation on the data. It allows the cache miss

Table 7.
IEEE floating-point data-manipulation instructions.

Opcode	Operands	Function
Fadd.siz	rd,rs1,rs2	F-P addition
Fdiv.siz		F-P division
Fmul.siz		F-P multiplication
Fsub.siz		F-P subtraction
Flt.siz	rd,rs1,rs2	Conversion into F-P
Fcmp.siz	rd,rs1,rs2	F-P comparison
Int.siz		Conversion into integer
Nint.siz		Conversion into integer rounded
Trnc.siz		Conversion into integer truncated

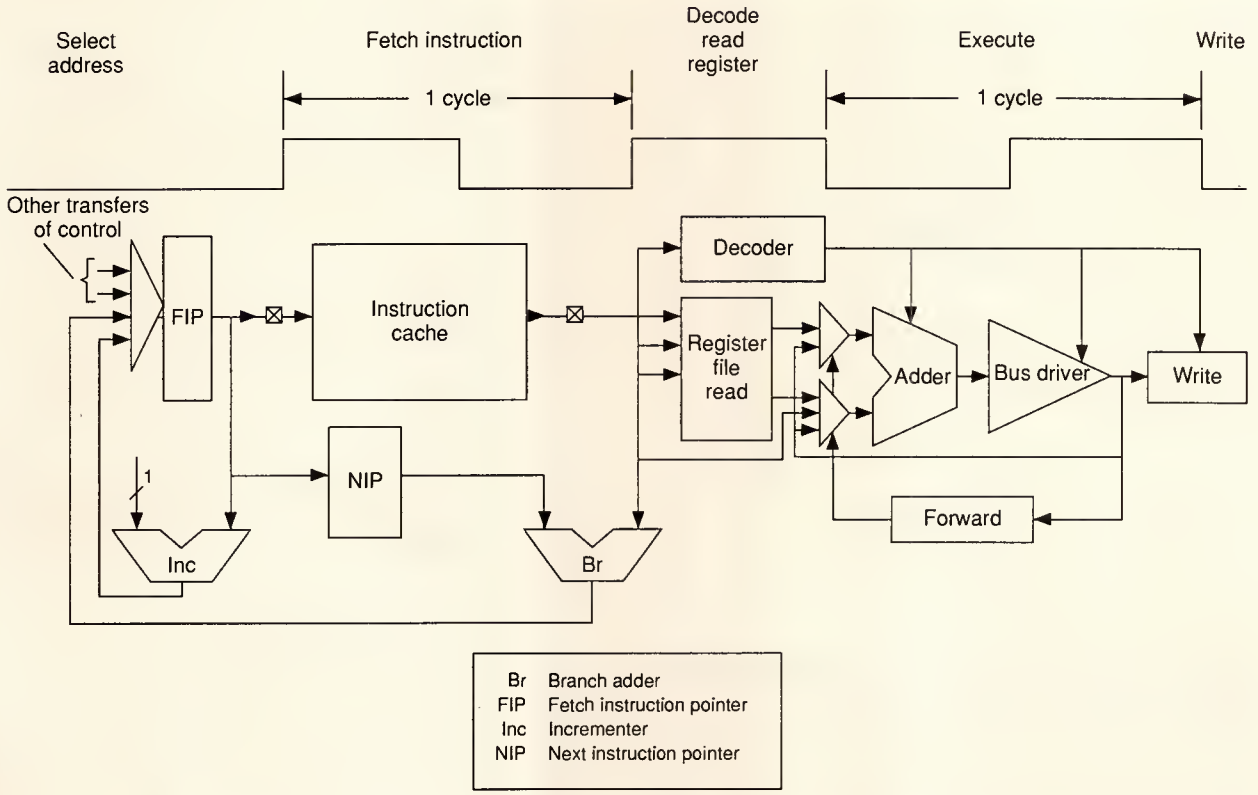


Figure 8. Sequencing in the master pipeline.

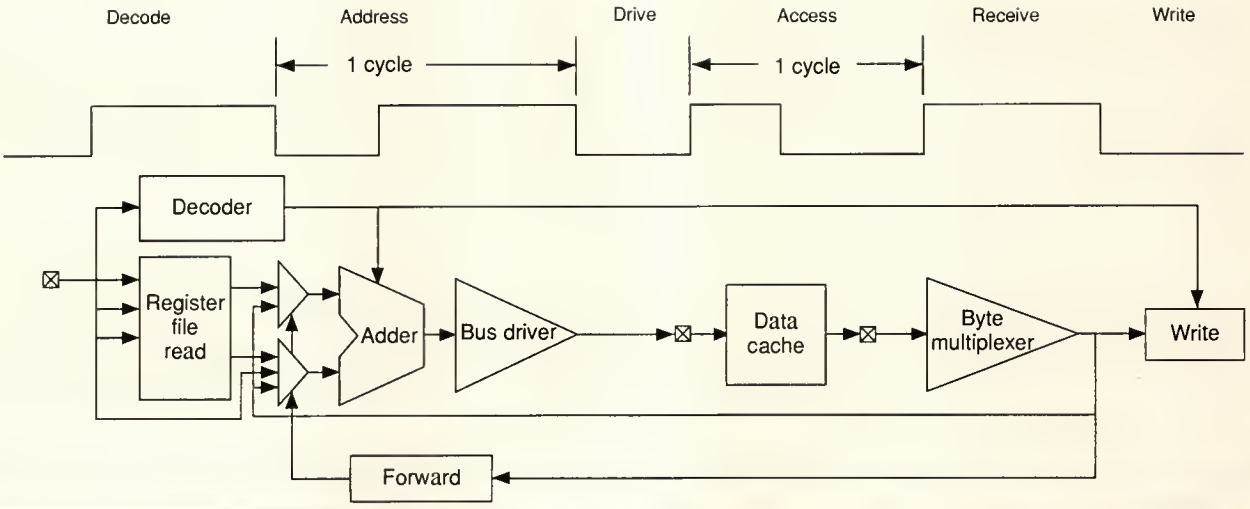


Figure 9. Memory unit pipeline.

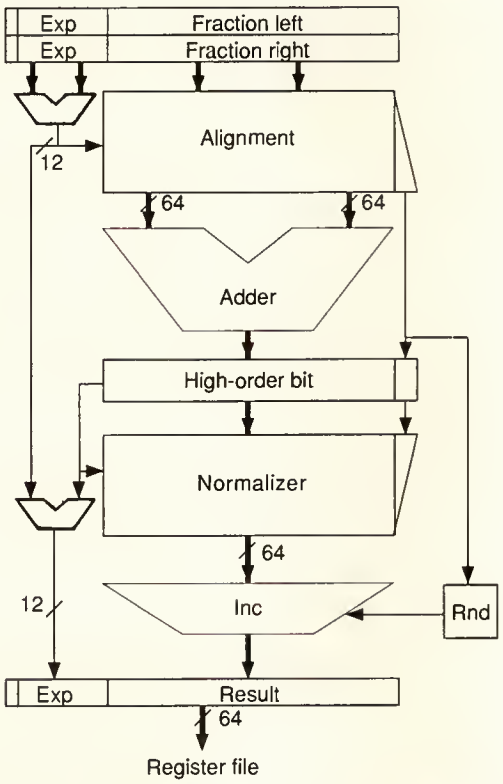


Figure 10. Floating-point addition pipeline.

time to bring in the data from main memory concurrently with the processing of the previous data, hiding most of the cache miss penalty

We felt the floating-point function units could not be placed close enough to the integer core to avoid impacting the register read and operand forwarding performance. Therefore we decided to set the floating-point units one-half cycle away from the integer core. These units incur the cost of an extra cycle of latency but ease the timing constraints on the processor. The floating-point adder pipeline takes four cycles, but the latency as seen at the register file is five cycles.

In the first cycle of any floating-point operation, the floating-point adder checks the source operands for reserved operands and traps to software if any occur. Software can complete the requirements in IEEE Std. 754 by implementing infinity, denormalized, and NaN arithmetic.

The floating-point adder is a 64-bit-wide, four-cycle pipeline that requires two register cycles to receive double-precision operands and return double-precision results. The first stage checks the operands, examines the exponent fields of the operands, and aligns the binary point of the smaller operand to that of the larger. The second stage performs the operation (addition, subtraction, comparison, or division). The third stage examines the resulting fraction and suitably realigns the binary point. The fourth stage performs any required rounding and delivers the result back to the register file. The second stage of the pipeline processes floating-point and integer division, which requires many cycles to form the quotient.

Figure 10 shows the floating-point addition pipeline. Here, operands from the register file expand into a canonical form while exponents (Exp) are compared and reserved operands are detected. The fraction with the smaller exponent aligns to that of the larger exponent. The sum (or difference) is computed, and the high-order bit of the result is found to allow normalization to occur. Finally, the rounded result moves back to the register file.

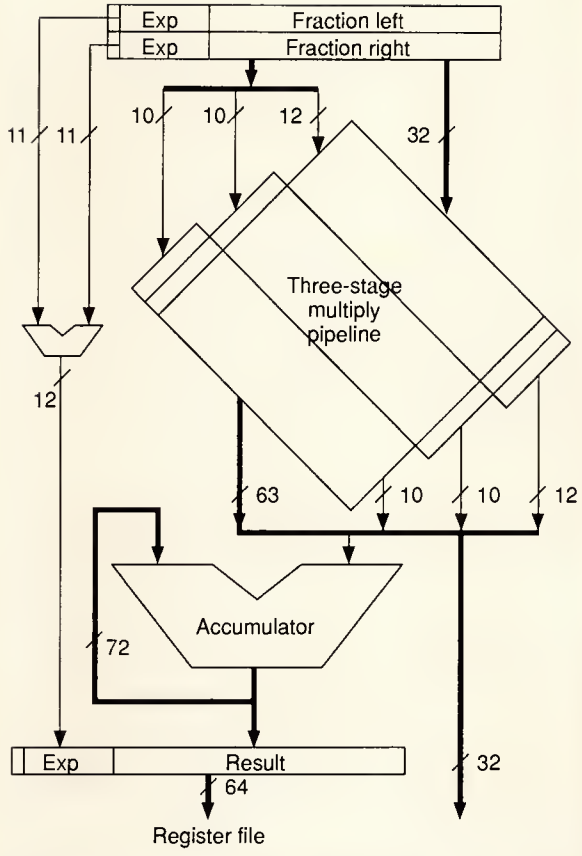


Figure 11. Multiplication pipeline.

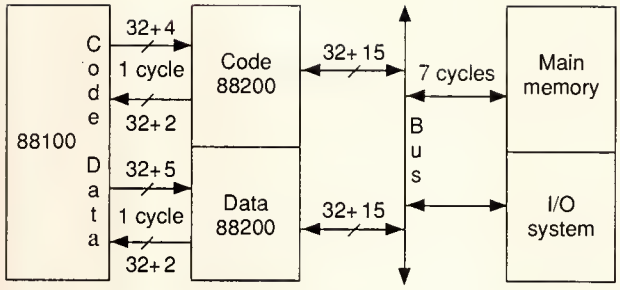


Figure 12. Relationship between the 88100 processor and two cache and memory management chips and their relationship to main memory and any associated I/O system.

The 88100 is the first microprocessor to contain a 32-bit × 32-bit array multiplier on the same chip as the integer unit. It implements fully pipelined integer and single-precision floating-point multiplication. The multiplication array must have already gone through three pipeline cycles to allow sufficient time for the data to fall through the array. The first stage multiplies one 32-bit operand by the low-order 12 bits of the other operand. Stage two continues the multiplication with the middle-order 10 bits. Stage three finishes the 32 × 32 multiplication with the high-order 10 bits. Floating-point multiplication continues with a partial product accumulator, which converts the high-order redundant form of the multiplication array into binary. Finally, the result is rounded.

Integer multiplication takes three cycles in the multiplier, and single-precision multiplication takes five, though both multiplications can issue each cycle. Double-precision operations require two register read cycles to obtain operands and two register write cycles to deliver results. The multiplication array requires four cycles to compute the partial products of double-precision operation and four cycles in the accumulator to add the partial products into a final product. It appears as a total latency of nine cycles. A new multiplication can begin four cycles after a double-precision multiplication starts. Mixed-mode multiplication requires two cycles in the array and two cycles in the accumulator; it can proceed at a two-cycle pipelined rate.

In Figure 11 we see the multiplication pipeline splitting the 32-bit × 32-bit multiplication array into three stages of 12, 10, and 10 bits. Binary bits "fall off" the right-hand side of the array, while redundant bits come off the left-hand side. Integer multiplication returns the lower 32 bits from the array, and single-precision multiplication uses the accumulator to resolve the redundant form from the multiplier. Mixed-mode and double-precision multiplications take several passes through the multiplication array and the accumulator to compute their results.

Cache design

All of this performance capability would go underutilized if the processor had been forced to wait several cycles on each memory access. A small instruction and/or data cache on the (already large) processor chip would suffer frequent cache misses and main memory access delays. By devoting an entire chip to support the memory bandwidth requirements, we could make a reasonably large cache memory and use two of these for the processor's two memory ports, as shown in Figure 12. The 88200 cache is a high-speed memory system that permits one-cycle access in a pipelined fashion.

A cache memory associates an address and some state with each container of data in a tag, or storage area for this association. By associating several locations with one address, we reduce the overhead of the tag memory. In addition, by transferring several words of data from the main memory system on one request, the latency of subsequent accesses can be reduced, and the effective bus bandwidth increased. The 88200 provides containers of 4 words, or one line.

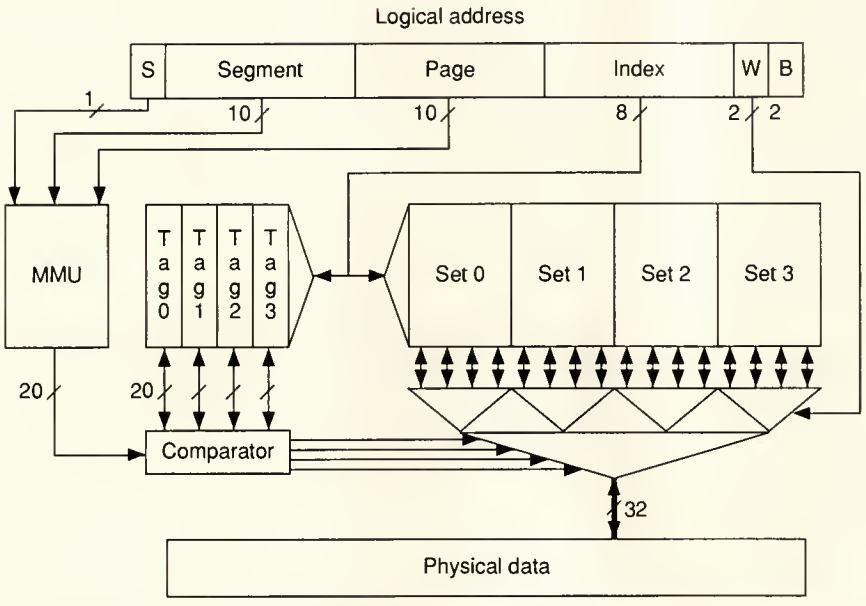


Figure 13. Logical address translation.

The 88200 is a four-way set-associative cache. Each access reads four tags and prepares access to four (partial) data lines. The 88200 compares the four accessed tags with the translated address to determine if any of the lines match the data requested by the processor.

As seen in Figure 13, an incoming logical address (and its space identifier, user U or supervisor S) accesses one word of physical data. The low-order 2 bits (B) of the address specify the byte displacement within a word, the next 2 more significant bits specify the word (W) within a line, and the next 8 more significant bits specify the line within the cache. This portion comes directly from the logical (untranslated) address supplied by the processor. The index bits select four tags and four candidate data words. The upper 20 logical address bits translate in the MMU concurrently with the access of the tags and data.

If any of the tags match the translated address, that word is connected to the read/write circuitry. By performing the cache access and the memory management translation concurrently, the cache hides the address translation time. This is a logically indexed, physical cache.

Cache size. The performance of any cache is primarily a function of its size. The various protocols, memory system parameters, and the bus performance secondarily affect performance. A bigger cache that holds more items has a lower probability that a reallocation of a line will displace data that will be needed shortly.

We obtained the memory cell from our 64K × 1, high-speed static RAM along with several of its designers and determined that we could afford 16 Kbytes of data storage in our integrated cache chip. The data store and the tag store total 155 Kbits. Available literature indicated that this amount of memory should provide a very efficient cache memory system. However, we decided that one cache size did not fit all requirements and allowed several cache chips to be cascaded for larger caches.

The 88200 services processor requests each cycle. When the 88100 sends an address to the 88200, the 88200 accesses its internal data, tag, and translation caches. It then determines if the access can be completed in the current cycle. All cache hits process in one cycle. When a data cache miss occurs, a line from the set of four is selected for replacement by means of a least recently used algorithm. If the line contains modified data, the processor writes it back to main memory before reading the requested data and inserting it into the cache line.

Memory management. The 88200 also performs memory management functions (mapping, relocation, protection). The MMU maps logical addresses (manipulated by the application) into physical addresses (assigned by the operating system). The MMU accomplishes the mapping by partitioning memory into 1,024 four-Mbyte segments, each of which contain 1,024 four-Kbyte pages.

The MMU translates a logical address by indexing through a two-level table. The user or supervisor root pointer control register holds the table's base address. The high-order 20 bits from this register and the high-order 10 bits of the logical address are concatenated to form the address of a segment table entry. This memory location contains the high-order 20 bits of the page table and access control information. Its high-order 20 bits and the next 10 untranslated bits are concatenated to form the address of the page table entry. This memory location contains the 20-bit physical page address and more access control information. The 20-bit physical page address and the 12 remaining untranslated address bits form the physical address of the request.

In addition to mapping logical addresses to physical addresses, the 88200 MMU monitors the access rights and the coherence strategy chosen, and it maintains a used and modified status for each page. Each page can be marked either as accessible only to the supervisor or

as write protected from the user. A cache inhibit bit, a global bit, and a write-through bit control memory coherence. Cache-inhibited memory locations must always be accessed in main memory and are allocated in the cache. Global pages are "snooped" to maintain cache and memory coherence.

Figure 14 shows the memory management table structure. The space identification bit specifies a supervisor or user access. The segment table bits from the logical address combine with the selected space register to form the address of the segment table entry. Next, the page bits of the logical address combine with the segment table entry to form the page table entry address. Finally, the index, word, and byte bits are combined with the page table entry bits to form the physical address.

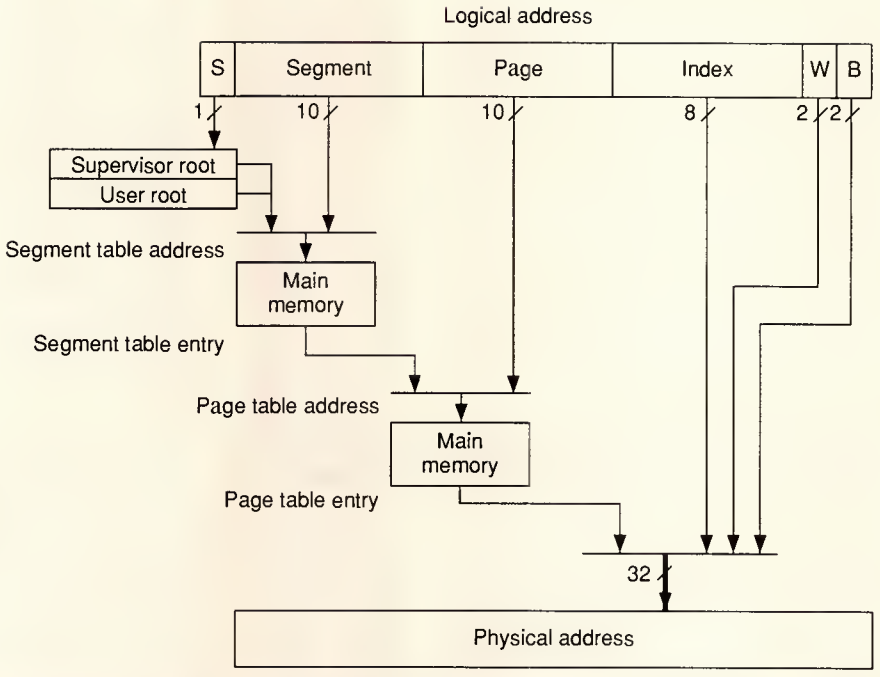


Figure 14. Structure of the memory management table.

We improved the performance of the MMU in the 88200 by having it keep a cache of translated pages called the page address translation cache.[18] This fully associative cache holds 56 of the most recent translations. Logical addresses stay in a content-addressable memory, or CAM (shown in Figure 15). In this entry, logical address hits cause the read of the associated RAM location with the translated address. A hardware-controlled table-walk mechanism (mentioned earlier) fills this translation buffer automatically. The CAM translation mechanism uses one-half cycle.

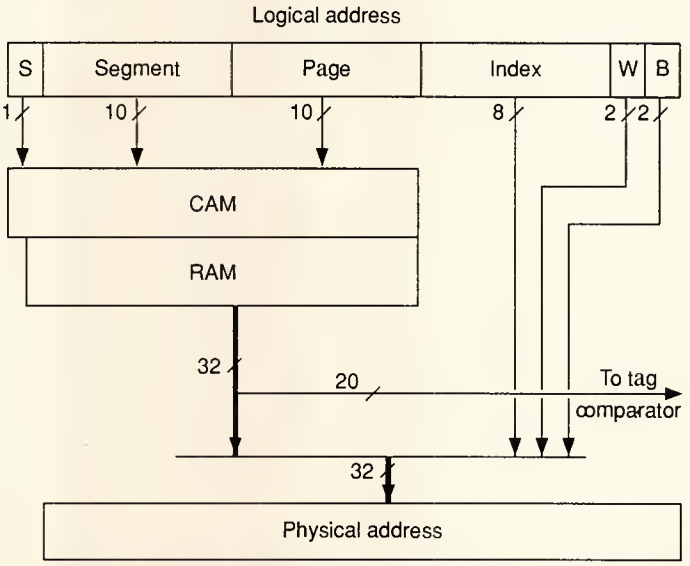


Figure 15. Content-addressable memory translation.

In addition to this page address translation cache, another software-managed translation mechanism, the block address translation cache (BATC), maps large memory areas such as the operating system, reentrant libraries, and databases (not shown in the figure). The BATC contains eight entries. Both of these translation mechanisms contain information concerning the rights of the process to access the data, and the caching protocol for the data.

Cache coherence. Stores from the processor can be written into the cache and forwarded to main memory at the same time in a policy known as write through. Alternatively, the data can be held in the cache and written back to main memory only when a line is reassigned on a cache miss in the policy called write

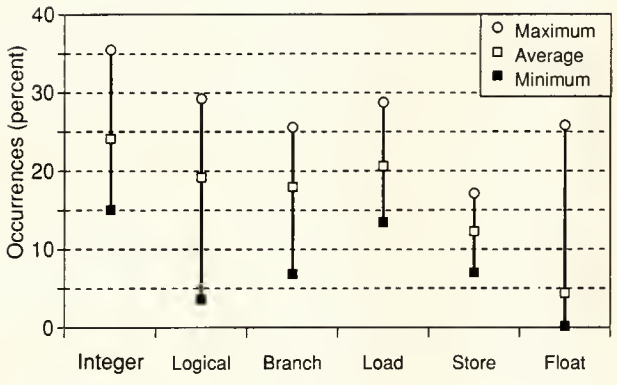


Figure 16. The 88100 instructions used by compilers.

back. Since a write-back cache memory can hold data differing from the data in main memory, the contents of the cache are inconsistent with the main memory. Even a write-through cache can become inconsistent when another party (direct memory access) modifies main memory.

We organized the 88200 cache by physical addresses to simplify explicit sharing of data between processes and, by happenstance, sharing of common libraries of code. When two logical addresses point to one physical address, we have an alias between these addresses. Since data is organized by its physical address, the 88200 will find any aliased logical addresses. By avoiding aliasing problems within and between processes, the cache does not need to be flushed when switching context from one process to another.

The 88200 solves the cache consistency problem by monitoring the memory bus on all address cycles. When the 88200 determines that an external device is requesting modified data within its cache, it intervenes to prevent use of any stale data in a process that has come to be known as "snooping the bus." The 88200 implements a modified form of the write-once protocol, commonly known as a Goodman protocol.[19] It implements two modifications: Addresses can be marked as nonsnooped, and pages can be marked as write through instead of write back.

Writing the data to main memory (the first time) establishes ownership of newly modified data. Since other caches are also snooping the bus, they invalidate any copy of this cache line. Subsequent writes to the line process without additional memory bus cycles. Ownership is revoked when another cache accesses the line. If the line has been modified only once, memory is consistent with the cache, and the cache simply removes its ownership. If the line has been modified with respect to memory, the 88200 aborts the external request, arbitrates for the bus, and updates main memory. The aborted request restarts and obtains the correct data.

Each cache chip contains a large number of pins devoted to servicing processor requests. We decided to limit the number of pins on the memory bus by multiplexing the address and data lines. We expected most systems would use dynamic RAMs that require an address quite some time before they can deliver data or require data to be written. The delay in between the address phase and the data phase is used to snoop the interested bystander caches. Thus, we felt multiplexing should cause only a small loss in performance. We amortized the access time of main memory by allowing the four words of data to cross the bus in four successive clock cycles.

Using the 88000 family

Several computer manufacturers use the 88000 family as one of their central processing engines. These implementations differ in the frequency of operation, main memory speed, and number of cache chips used. The following presents some data on the use of the instruction set by the available compilers and the efficiency of the cache and memory systems. Note that the compiled code used in this study is at least 18 months old and that the compilers now produce significantly higher performance.

Instruction statistics. We analyzed the first-generation 88000 compilers to determine how well they used the instruction set of the machine. We examined a moderate number of benchmarks (10) in detail on the 88100 simulator. Figure 16 displays the average number of instructions of various classes executed by these benchmarks. These averages are in accordance with several other studies on instruction-set statistics. The graph shows how the first-generation compilers use the instruction set of the 88100 processor. Logical instructions accomplish three purposes: logical operations, data movement, and creation of the high-order 16 bits of 32-bit quantities.

We found logical instructions to be more frequently used in the 88100 than in other studies. The 88000 compilers use the logical Or instruction as a register-to-register move instruction. Absolute memory-addressing references are constructed by using (again) the logical Or instruction to "paste" a 16-bit constant into the high-order 16 bits of a register. This value is added to the low-order 16 bits contained within a load or a store instruction, forming a full 32-bit address.

Instruction throughput. The performance of a system depends on the efficiency of the processor, its caches, and its compilers. We analyzed a large number of applications and benchmarks to determine processor efficiency when running compiled high-level-language code. The processor averaged 1.324 cycles per instruction on integer applications with a perfect memory

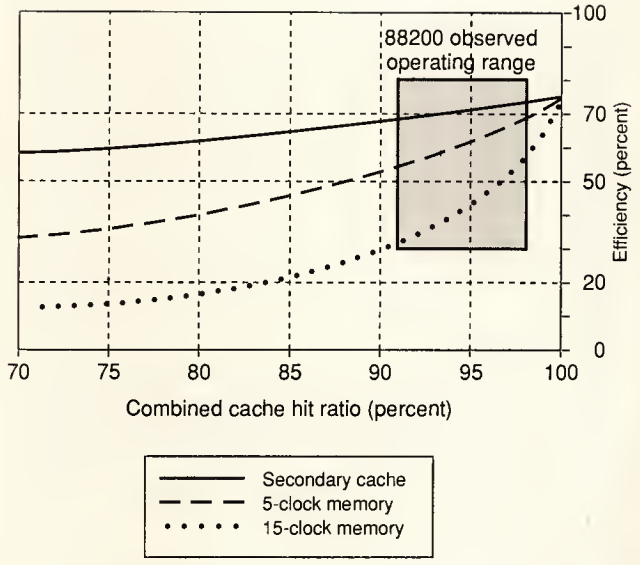


Figure 17. Performance range of an 88100/200 system based on the access time of the main memory system.

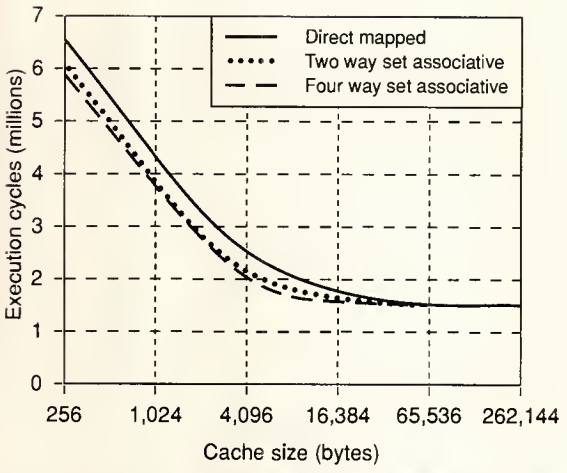


Figure 18. Effectiveness of the 88200 caches on a program with above average code and data locality. The four-way set-associative design reaches minimum execution cycles earlier than a direct-mapped cache.

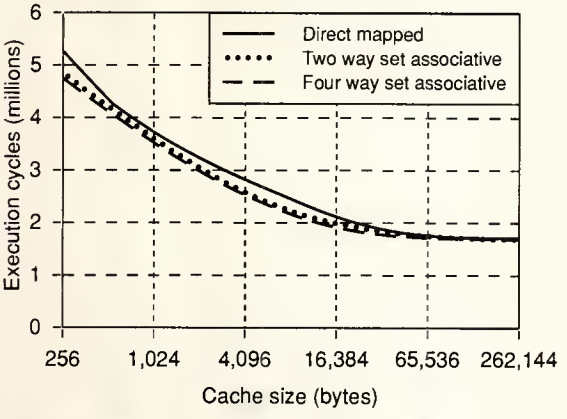


Figure 19. Effectiveness of set-associative caches on a program with average code and data locality. A 30-Kbyte data structure in this program is reinitialized for each function being compiled.

system. This rate corresponds to 0.755 instructions per cycle. An imperfect memory system can only degrade the system performance.

Figure 17 relates the performance levels achievable with several different memory systems across the combined code and data cache hit ratio. We've seen a single pair of 88200s with a hit ratio on average applications between 92 percent and 96 percent. By cascading several 88200s on each processor port, we can improve the hit ratios. Cascading improves the performance of systems with slower memories much more than it would a system using fast secondary caches. Compiler improvements can increase the baseline efficiency of the processor.

Cache performance. Cache memories reduce main memory latency when accesses display spatial and temporal locality. Small benchmarks that fit into the cache yield performance levels that are not sustainable on average applications.

Figures 18 through 20 contrast small, medium, and large applications over a range of cache sizes and organizations.

The small program in Figure 18 is the public-domain version of the XLisp interpreter running the small, heavily recursive Tak benchmark. It demonstrates good spatial and temporal locality while executing this benchmark.

The medium benchmark in Figure 19 is an 88000 version of the PCC compiler compiling the Dhrystone 1.1 benchmark. This application demonstrates good instruction locality, but the reinitialization of a single data structure destroys the data cache between function compilations.

The large program in Figure 20 on the next page is a transaction-processing benchmark applied to a large database. This application demonstrates poor locality on both instruction and data references, but it shows how the processor, cache, and memory system work in harmony. Higher performance main memory requires less buffering by the cache than lower performance main memory, or higher hit ratio caches must compensate for lower performance main memories. The flattening of the execution cycles near a 1/4-megabyte cache size may be an artifact of the length of the instruction trace.

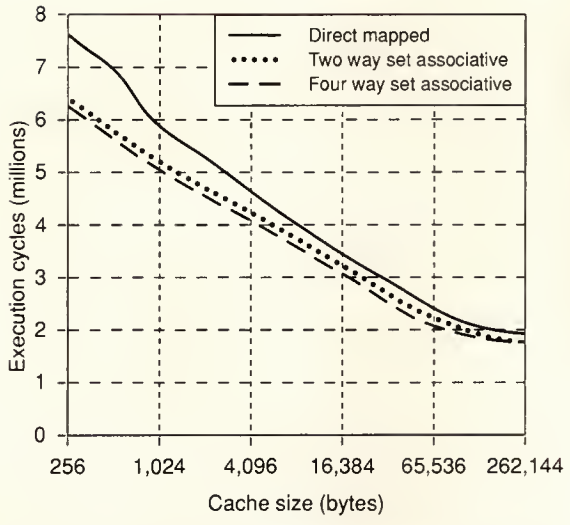


Figure 20. Effectiveness of set-associative caches on a large program. This program manipulates a large database (around 500 pages) and is itself a large program (around 100 pages).

Transient cache analysis. The miss ratio of a cache memory is not a constant. Transients exist when the program starts up , and additional transients show up as the program migrates from one chore to another. Figure 21 displays the miss ratio of both code and data caches of an 88000 system with a pair of 88200s. The application is a circuit simulator with 10 circuits sequentially undergoing simulation. The first few circuits are small and have good cache behavior, but the larger circuits display very large code miss rates. This benchmark executes 263,470,685 instructions, 75,105,619 loads, and 41,001,290 stores.

The figure shows that programs change their hit rates as they move from one work load to another. Some sections of a work load fit neatly into the cache, while others do not. Although the overall miss rate of the code section of this program is around 2 percent, almost half the time occurs with a miss rate of 7 percent to 9 percent.

Several of the minor areas of the 88100 and 88200 design were not as successful as desired, and several of the decisions proved to be more successful than anticipated.

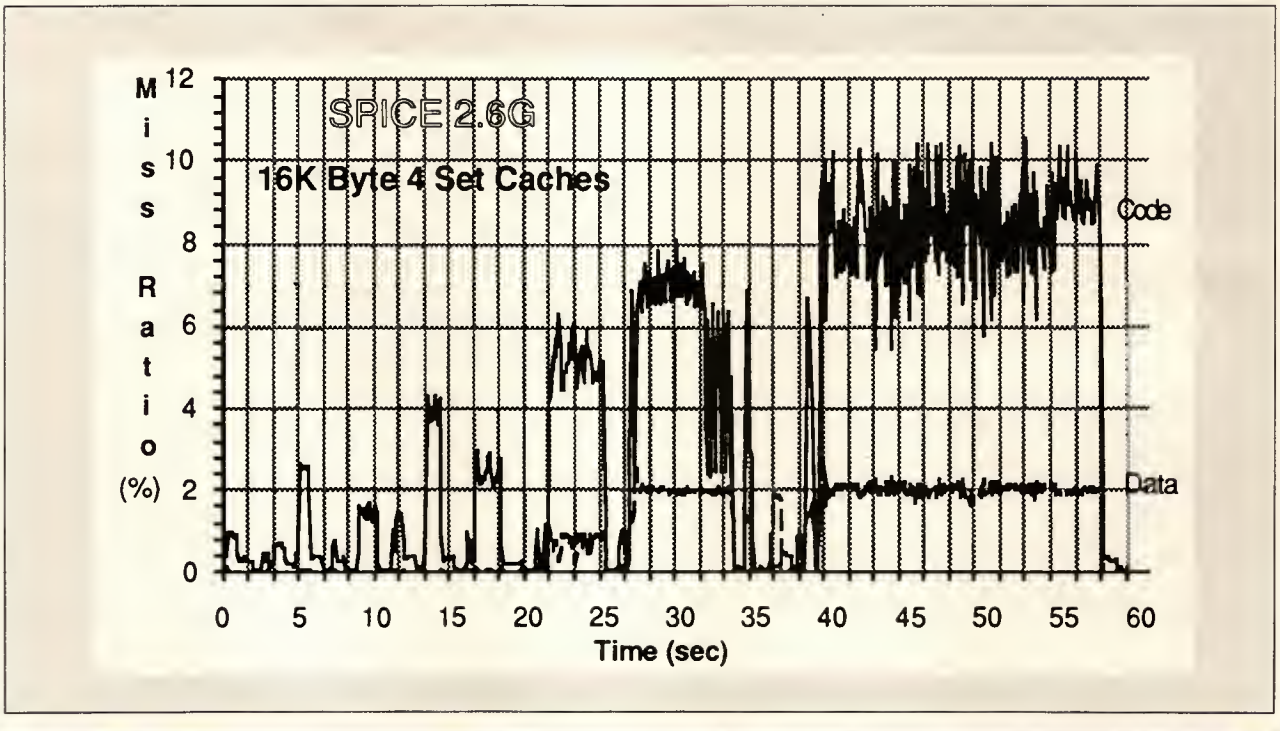


Figure 21. Hit rate changes.

READER-SERVICE CARD

IEEE MICRO

June 1990 issue (card void after December 1990)

Name _____

Title _____

Company _____

Address _____

City _____ State_____ ZIP _____

Country _____ Phone (_____) _____

Please send (Circle those you want)

201 Publications catalog
202 Membership information
203 Student membership information
204 Senior membership information
205 *IEEE Micro* subscription information

Reader Interest
(Add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles and departments):

150 151 152 177 178 179
153 154 155 180 181 182
156 157 158 183 184 185

159 160 161 186 187 188
162 163 164 189 190 191
165 166 167 206 207 208

168 169 170 209 210 211
171 172 173 212 213 214
174 175 176 215 216 217

Product Information 1
(Circle the numbers to receive product information)

1	21	41	61	81	101	121	141
2	22	42	62	82	102	122	142
3	23	43	63	83	103	123	143
4	24	44	64	84	104	124	144
5	25	45	65	85	105	125	145
6	26	46	66	86	106	126	146
7	27	47	67	87	107	127	147
8	28	48	68	88	108	128	148
9	29	49	69	89	109	129	149
10	30	50	70	90	110	130	—
11	31	51	71	91	111	131	—
12	32	52	72	92	112	132	192
13	33	53	73	93	113	133	193
14	34	54	74	94	114	134	194
15	35	55	75	95	115	135	195
16	36	56	76	96	116	136	196
17	37	57	77	97	117	137	197
18	38	58	78	98	118	138	198
19	39	59	79	99	119	139	199
20	40	60	80	100	120	140	200

READER-SERVICE CARD

IEEE MICRO

June 1990 issue (card void after December 1990)

Name _____

Title _____

Company _____

Address _____

City _____ State_____ ZIP _____

Country _____ Phone (_____) _____

Please send (Circle those you want)

201 Publications catalog
202 Membership information
203 Student membership information
204 Senior membership information
205 *IEEE Micro* subscription information

Reader Interest
(Add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles and departments):

150 151 152 177 178 179
153 154 155 180 181 182
156 157 158 183 184 185

159 160 161 186 187 188
162 163 164 189 190 191
165 166 167 206 207 208

168 169 170 209 210 211
171 172 173 212 213 214
174 175 176 215 216 217

Product Information 2
(Circle the numbers to receive product information)

1	21	41	61	81	101	121	141
2	22	42	62	82	102	122	142
3	23	43	63	83	103	123	143
4	24	44	64	84	104	124	144
5	25	45	65	85	105	125	145
6	26	46	66	86	106	126	146
7	27	47	67	87	107	127	147
8	28	48	68	88	108	128	148
9	29	49	69	89	109	129	149
10	30	50	70	90	110	130	—
11	31	51	71	91	111	131	—
12	32	52	72	92	112	132	192
13	33	53	73	93	113	133	193
14	34	54	74	94	114	134	194
15	35	55	75	95	115	135	195
16	36	56	76	96	116	136	196
17	37	57	77	97	117	137	197
18	38	58	78	98	118	138	198
19	39	59	79	99	119	139	199
20	40	60	80	100	120	140	200

SUBSCRIPTION CARD

SUBSCRIBE TO IEEE MICRO

All the facts about today's chips and systems

❑YES, sign me up!

If you are a member of the Computer Society or any other IEEE society, pay the member rate of only **$19** for a year's subscription (six issues).

Society: _____ IEEE membership no.: _____

Society members: Subscriptions are annualized. For orders submitted March through August, pay half the full-year rate $9.50 for three bimonthly issues.

❑YES, sign me up!

If you are a member of ACM, ACS, BCS, IEE (UK), IEEE (but not a member of an IEEE society), IECEJ, IPSJ, NSPE, SCS, or other qualified professional technical society, pay the sister-society rate of only **$35** for a year's subscription (six issues).

Organization: _____ Membership no.: _____

Full Signature *Date*

Name

Street

City

State/Country *ZIP/Postal Code*

❑ Payment enclosed

❑ Charge to ❑ Visa ❑ MasterCard ❑ American Express

Charge-card number

Expiration date

Month *Year*

Prices valid through 12/31/90
6/90 Micro

Change orders also taken by phone:
(714) 821-8380 8a.m. to 5 p.m. Pacific time
Circulation Dept.
PO Box 3014
Los Alamitos, CA 90720-1264

Editorial comments

I liked: _____

_____

_____

_____

_____

I disliked: _____

_____

_____

I would like to see: _____

_____

_____

Reviewers needed. *If interested, send professional data to Joe Hootman, EE Dept., University of North Dakota, PO Box 7165, Grand Forks, ND 58202.*

PO Box is for reader-service cards only.

PLACE POSTAGE HERE

IEEE Micro
Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

Editorial comments

I liked: _____

_____

_____

_____

_____

I disliked: _____

_____

_____

I would like to see: _____

_____

_____

For reader-service inquiries, see other side.

Reviewers needed. *If interested, send professional data to Joe Hootman, EE Dept., University of North Dakota, PO Box 7165, Grand Forks, ND 58202.*

PO Box is for reader-service cards only.

PLACE POSTAGE HERE

IEEE Micro
Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 38 LOS ALAMITOS, CA

POSTAGE WILL BE PAID BY ADDRESSEE

IEEE COMPUTER SOCIETY
Circulation Dept.
PO Box 3014
Los Alamitos, CA 90720-9804
USA

Since applications are not very amenable to cache memory technology, the 88200 treats all memory references equally with regard to cache allocation. Real-time systems may want time-critical data to remain in the cache at the expense of normal data. Transaction-processing applications wander around large programs and databases, carrying small structured units of data and performing minimal processing for each memory request. Both of these applications are characterized by a lack of temporal or spatial locality.

Other applications are amenable to cache memory technology only when the cache is of at least a certain size. Many of these programs are large numerics benchmarks, in which linear access is made to an array that is larger than the cache (maybe any reasonable cache). The performance of these programs is characterized more by the throughput of the memory system than by the speed of the processor. Indeed, supercomputers have increased memory bandwidth in direct proportion to the throughput of their function units. The usable bandwidth of the memory bus on the 88200 is somewhat disappointing.

The 88100 and 88200 implement a three-cycle memory pipeline. We've found it difficult to hide the extra cycle of latency on a load instruction with code scheduling. Very high speed static-RAM designs permit two-cycle load access. Conversely, the integrated design of the 88200 allows partial word writes to occur in one cycle.

Placing the MMU in parallel with the cache unit allows the address translation time to be hidden from cache memory accesses. This step requires that the data cache be indexed by logical address bits and checked against physical address bits. When the cache size is greater than the page size, the cache must be set associative in nature. Luckily, in CMOS VLSI chips, this delay is negligible compared to driving the data pins.

Our fast integer multiplication directly resulted from the inclusion of floating-point arithmetic on the processor. The speed of this instruction allows the efficient manipulation of multidimensional array indexing and is surprisingly effective in increasing performance of many applications.

Conversely, our double-precision floating-point performance does not stand up to our integer and single-precision performance levels. The register file contains only three 32-bit ports, one shared between results and stores. In double-precision code this resource is very hard to schedule among loads, operations, and stores. The length of the floating-point pipelines causes some additional loss of ultimate performance.

Acknowledgments

I am greatly indebted to Carl Dobbs, Janet Shooch, and Yoav Talgam without whom it would have been impossible to finish the 88000 architecture, the 88100 processor, and the 88200 cache and memory management chips. Additionally, I am grateful to Silicon Compiler Systems for the GDT tools used in the design, layout, and simulation of the 88100 processor and for coimplementing the 88200 cache and memory management chip.

References

1. D. MacGregor, D. Mothersole, and B. Moyer, "The Motorola MC68020," *IEEE Micro*, Vol. 4, No. 4, Aug. 1984, pg. 116.
2. J. Hennessy and M. Horowitz, "An Overview of the MIPS-X-MP Project," Computer Systems Laboratory, Stanford University, Stanford, Calif., Tech. Report 86-300, pg. 2.
3. J. Davidson and R. Vaughan, "The Effect of Instruction Set Complexity on Program Size and Memory Performance," ACM, New York, pg. 62.
4. J. Emer and D. Clark, "A Characterization of Processor Performance in the VAX-11/780," Table 3, pg. 306.
5. D. MacGregor and J. Rubenstein: "A Performance Analysis of MC68020-Based Systems," *IEEE Micro*, Vol. 5, No. 6, Dec. 1985, pp. 50-70.
6. M. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge, Mass., 1985, pg. 44.
7. S. Przybylski et al., "Organization and VLSI Implementation of MIPS," Tech. report, Computer Systems Laboratory, Stanford University.
8. A. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM*, Vol. 21, No. 3, Mar. 1978, pp. 237-246.
9. J. Thornton, "Parallel Operations in the Control Data 6600," *AFIPS Conf. Proc. FJCC Pt. II*, Reston, Va., Vol. 26, 1964, pp. 33-40.
10. G. Bell and A. Newel, *Computer Structures: Readings and Examples*. McGraw-Hill Book Co., New York, 1971, pp. 489-496.
11. N. Joppi and D. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proc. Third Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press, Los Alamitos, Calif., Apr. 1989, pp. 272-282.
12. J. Liptay, "The Model 85 Buffer Storage," *IBM Systems J.*, Vol. 7, No. 1, 1968.
13. A. Smith: "CPU Cache Memories," *ACM Computing Surveys*, New York, Vol. 14, No. 3, Sept. 1983, pp. 473-530.
14. *IBM System/370 Principles of Operation, GA22—0644*, (5th ed.), IBM Corp., 1976.

15. Digital Equipment Co., *VAX Hardware Handbook*, Marlboro, Mass., 1978.

16. D. Patterson and D. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News*, ACM SIGArch, Vol. 8, No. 6, Oct. 1980, pp. 25-33.

17. *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, IEEE CS Press, 1985.

18. F. Lee, "Study of 'Look Aside' Memory," *IEEE Trans. Computers*, Vol. 18, No. 11, Nov. 1960.

19. J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *10th Int'l Symp. on Computer Architecture*, 1983, pp. 124-131.

Mitch Alsup, a computer architect, led the design of the 88000 family of microprocessors. His work included the definition of the instruction set and the processor-to-cache bus interface, function unit partitioning of the 88100 processor, and circuit design and layout of the multiplier array. He currently works on advanced superscalar implementation technology. Other technical interests include compiler optimizations, high-performance memory systems, and VLSI circuit design.

Alsup received the BSEE degree from Carnegie Mellon University.

Questions concerning this article may be addressed to the author at Motorola, 6501 William Cannon Drive West, Austin, TX 78735-8598.

IEEE COMPUTER SOCIETY

IEEE COMPUTER SOCIETY
Membership / Subscription Application

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

BENEFITS

Computer
Computer comes automatically with membership. Written, reviewed, and refereed by experts, it features survey and tutorial articles covering the entire computer field, and departments such as new products, new product reviews, standards, and a reader forum called "The Open Channel." (monthly).

Technical Committees
Participate in one or more of our 33 technical committees — networks of professionals with common interests in specialty areas within computer hardware, software, and applications.

Standards Working Groups
Participate in the development of the more than 100 standards projects currently sponsored by the society in such diverse areas as software engineering, local area networks, microprocessor buses, design automation, programming languages, and standards definitions.

Computer Society Press Books
Receive discounts of up to 50% on over 600 titles covering a broad spectrum of computer science topics such as networking, communications, advanced systems, image processing, security, artificial intelligence, and design automation. Over 60 new titles are published annually.

Conferences and Tutorials
Choose from more than 100 conferences annually, ranging from large industry-oriented conferences replete with exhibits to small, highly interactive workshops. Members receive special low rates.

Schedule of Fees

To join: see item 1, 2, or 3.
To subscribe: see item 4.

Membership dues and periodical subscriptions are annualized to, and expire on, December 31. Choose full- or half-year rate schedules depending on date of receipt by the Computer Society as indicated below.

	Half Year Mar 1-Aug 31	Full Year Sept 1-Feb 28
1 I don't belong to the IEEE and I want to join just the Computer Society	☐ $23.50	☐ $47.00
2 I don't belong to the IEEE and I want to join both the Computer Society and the IEEE*		
I reside in Region 1-6 (United States)	☐ $47.50	☐ $95.00
I reside in Region 7 (Canada)	☐ $43.50	☐ $87.00
I reside in Region 8 (Europe, Africa, or the Middle East)	☐ $43.00	☐ $86.00
I reside in Region 9 (Latin America)	☐ $39.50	☐ $79.00
I reside in Region 10 (Asia and Pacific)	☐ $38.50	☐ $77.00
3 I already belong to the IEEE and I want to join the Computer Society.	☐ $ 9.00	☐ $18.00

*ACM members who join both IEEE and the Computer Society may deduct $5 off the full-year rate; $2.50 off the half-year rate.

IEEE Member Number _____

4 OPTIONAL PERIODICALS for new or current members

	issues per year		
IEEE Computer Graphics and Applications (3061)	6	☐ $10.00	☐ $20.00
IEEE Design and Test (3111)	6	☐ $10.50	☐ $21.00
IEEE Expert (3151)	6	☐ $ 9.00	☐ $18.00
IEEE Micro (3071)	6	☐ $ 9.50	☐ $19.00
IEEE Software (3121)	6	☐ $10.00	☐ $20.00
Transactions on Computers (1161)	12	☐ $10.00	☐ $20.00
Transactions on Knowledge and Data Engineering (1471)	4	☐ $ 5.00	☐ $10.00
Transactions on Parallel and Distributed Systems (1501)	4	☐ $ 5.50	☐ $11.00
Transactions on Pattern Anaysis and Machine Intelligence (1351)	12	☐ $10.00	☐ $20.00
Transactions on Software Engineering (1171)	12	☐ $10.00	☐ $20.00

Total amount remitted with this application $ _____

☐ Checks are accepted in Belgian, British, German, Swiss, Japanese, or U.S. currencies. U.S. checks must be drawn on a U.S. bank.

PRICES EXPIRE 12/31/90

☐ Visa ☐ Master Card ☐ American Express ☐ Eurocard

Charge Card Number _____ Exp. Date (Mo. / Yr.) _____

I hereby make application for Computer Society membership and, if elected, will be governed by IEEE's and the society's constitutions, bylaws, and statements of policies and procedures.

MAILING ADDRESS

Full signature _____ Date _____

First name _____ Middle initial(s) _____ Last name _____

Street address _____ City _____ State/Country _____ Zip _____

EDUCATION (highest level completed) Course _____ Degrees received _____ Date _____

Name of educational institution _____ Date of birth _____ ☐ Male ☐ Female

OCCUPATION Title or Position _____

Firm name _____

Firm address _____

City _____ State/Country _____ Zip _____

REFERENCE (an IEEE member; if unknown, a managerial person who knows you professionally)

Name (print in full) _____ IEEE Member No. (if applicable) _____

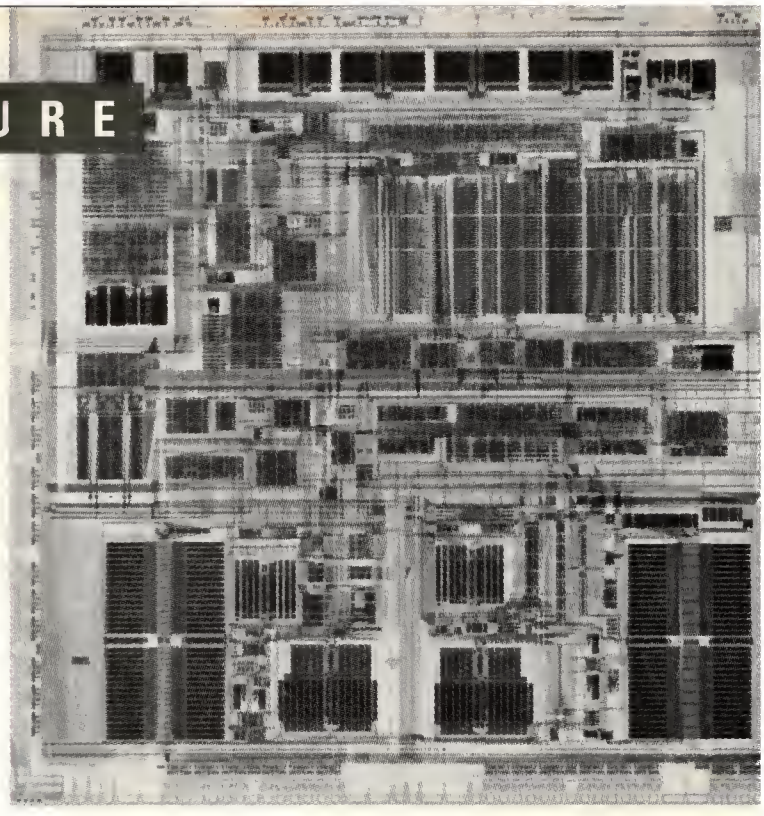
Street address _____

City _____ State/Country _____ Zip _____

Return to: IEEE Computer Society, 10662 Los Vaqueros Circle, P.O. Box 3014 Los Alamitos, CA 90720–1264 USA.
Residents of Europe mail to: IEEE Computer Society, 13, Avenue de l'Aquilon, B-1200, Brussels, BELGIUM.
Asian / Pacific residents mail to: IEEE Computer Society, Ooshima Building, 2-19-1 Minami-Aoyama, Minato-ku, Tokyo 107 JAPAN.

MICRO 06/90

SPECIAL FEATURE



The Gmicro/300 32-Bit Microprocessor

The Gmicro/300 is a high-end microprocessor based on the TRON architecture specification. In contrast to other RISC or CISC chips, it executes an instruction with a memory operand and a register operand in one clock cycle. The separate cache memories improve performance more than 13.8 percent.

Takeshi Kitahara
Taizo Satoh

Fujitsu Limited

The 32-bit Gmicro/300 microprocessor is based on the TRON architecture specification.[1,2] Plans call for this high-end, 20/25-MHz chip to be used as a workstation engine, personal computer, and office processor. Its performance in the Dhrystone benchmark program peaked at 14 million instructions per second during 20-MHz operation and 17 MIPS during 25-MHz operation. The chip basically supports 135 instructions. Table 1 lists additional details.

To improve performance, the Gmicro/300 includes a hardwired circuit for one-cycle execution of a simple arithmetic or logical operation. A microprogram executes complex instructions such as string operations and queue manipulations, and complex functions such as exceptions, interruptions, and dynamic address translations, or DATs. The microprogram and common execution circuits execute a DAT that begins when a miss occurs as an address is to be translated to the TLB unit (address translation lookaside buffer). We didn't want to provide exclusive hardware such as a sequencer for the DAT, so we must execute the DAT with the microprogram and the execution unit. We, however, realize that it is too difficult to implement in this way, because some current instruction function codes must be saved in internal latches before the DAT operation. Also, the codes must be restored afterward.

The basic cycle of the Gmicro/300 external bus completes in two clock cycles with a maximum transfer speed at 25 MHz of 80 million bytes per second. The bus also implements a burst transfer mode.[3]

Figure 1 shows the block diagram of the Gmicro/300. Its 11 units include the external bus interface, operand cache, operand cache tag, operand TLB,

0272-1732/90/0600-0068$1.00 © 1990 IEEE

Table 1.
Gmicro/300 features.

Item	Description	Item	Description
Clock rate	20 MHz, 25 MHz	Internal cache	Write through, physical
Performance	14 MIPS, 17 MIPS	Instruction	2 Kbytes
Total instructions	113 + 22 (coprocessor)	Operand	2 Kbytes
General registers	32 bits × 16		
		Bus snoop	Implemented
Address space		Bus cycle	Two clock cycles
Logical	4 Gbytes	Burst transfer	80 Mbytes/s (maximum at 25 MHz)
Physical	4 Gbytes		
Control*	4 Gbytes		
Protection	Four-level ring	Power	Typically 3 watts
		Transistors	About 900,000
Address translation		Chip size	15.98 × 15.98 sq mm
Instruction TLB	64 entries	Process	1.0-micrometer CMOS, three metal layers
Operand TLB	64 entries		

*Part of the control space exists outside the chip.

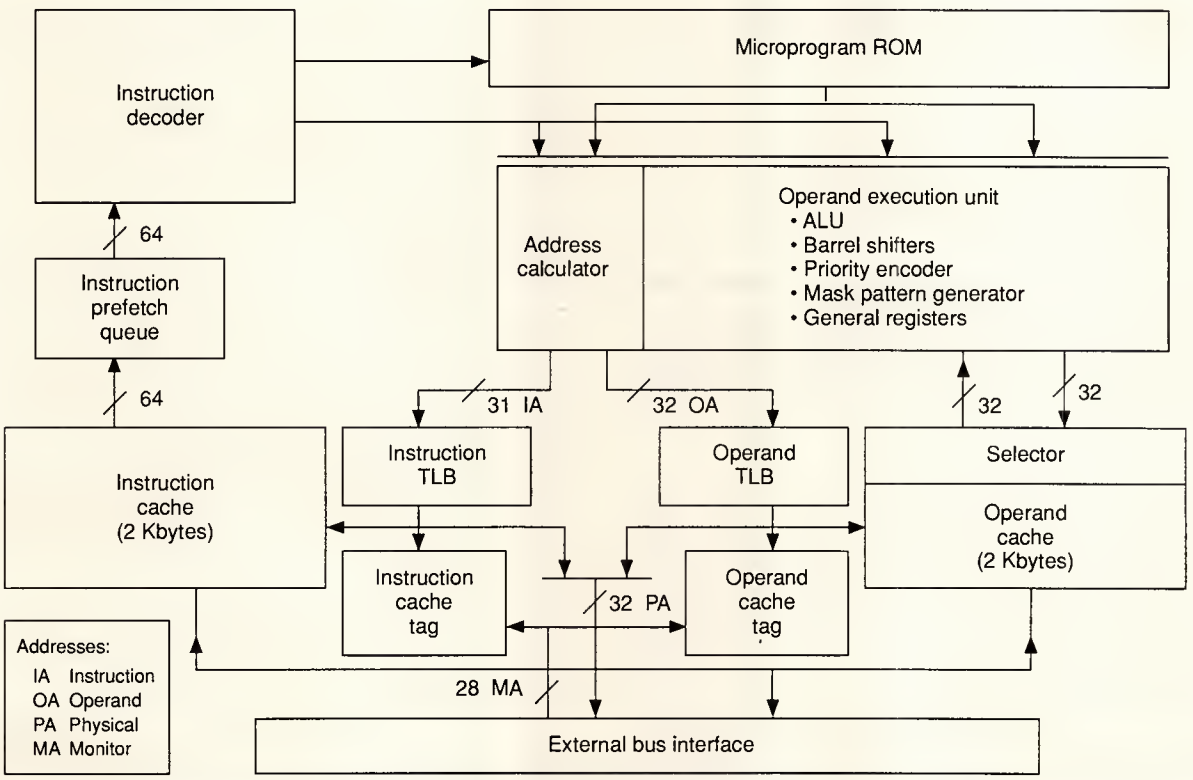


Figure 1. Block diagram of the Gmicro/300 microprocessor.

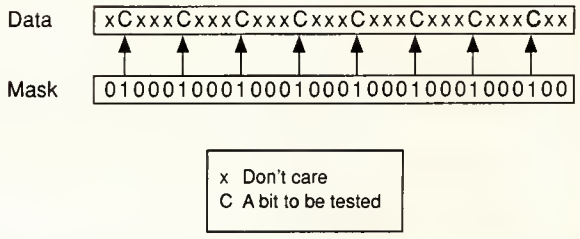


Figure 2. The MTST instruction, which uses condition flags L (all Cs set to 1) and Z (all Cs set to 0).

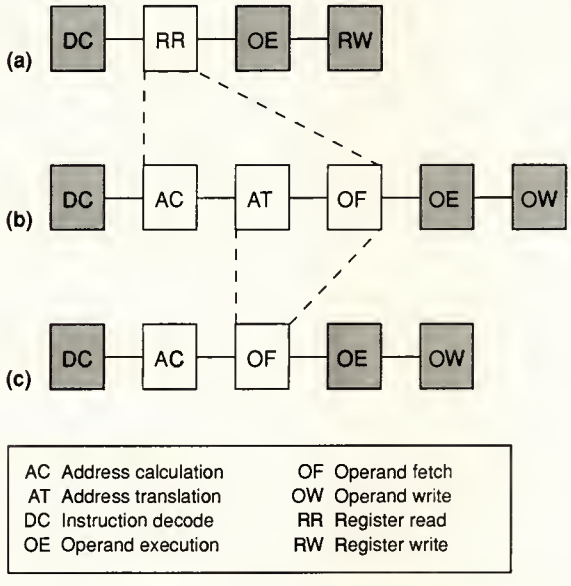


Figure 3. Pipeline structure for register-register (a), memory-register with address translation (b), and Gmicro/300 memory-register with address translation (c) operations.

instruction cache, instruction cache tag, instruction TLB, instruction prefetch queue, instruction decoder, microprogram, and operand execution unit. The operand execution unit includes an address adder that is separate from the ALU (arithmetic logic unit). The address adder can calculate operand addresses simultaneously with operand arithmetic.

Some Gmicro/300 instructions are useful for office applications and operating systems. One of our targets is office applications in which Cobol programs are widely used. In these programs signed decimal operations are indispensable. A signed decimal has two fields, the absolute value and the sign. When adding two signed decimals, we must consider each sign field and decide on the operation of the absolute values. Typically, a microprocessor checks the sign fields with a compare instruction, jumps with a conditional branch instruction, and lastly executes the absolute values. The Gmicro/300 supports signed decimal instructions that manage these operations internally.[4]

Some operating systems include tables for demand-paging management. One of these tables contains several sets of several bits that indicate free pages and modified pages. The sets search for and load a page from an external storage device or write a page to an external one. When the operating system searches a free page for a new requirement, it must test the bits in these bit fields. To do this, earlier architectures required the use of single-bit test instructions and conditional branch instructions, or some logical operations and a conditional branch instruction. The Gmicro/300 supports an MTST instruction that tests bits in data against the positions of bits set into 1s in a mask, and reflects the results in conditional flags (Figure 2). So, only two instructions, the MTST instruction and a conditional branch instruction, manage the same function.

Pipeline structure

In the Gmicro/300 the CPU operations determine the pipeline structure. Since the operations consist only of register-to-register operations, a four-stage pipeline is needed, as generally is the case in RISC (reduced instruction-set computing) architectures. Figure 3a shows the four stages of instruction decoding (DC), register read (RR), operand execution (OE), and register write (RW).

In the TRON architecture specification, instructions can include memory operands. If a CPU tries to execute a memory-to-register operation (Figure 3b,c) in the four-stage pipeline, it must calculate a memory address (AC) in the register read stage. It then must fetch a memory operand (OF stage) and execute it in the operand execution stage. The CPU requires at least two clock cycles to complete the operation because it must manage two functions in the operand execution stage.

In CISC (complex instruction-set computing) architectures, which have comparatively few general-purpose registers, memory operands tend to be used more often than in a register-based architecture. Total performance depends upon the operation speed of the memory operands in an architecture with few general-purpose registers.

Table 2 compares execution clock cycles required for a register-to-memory operation in RISC processors such as the Sun Microsystems, Inc. Sparc and the Intel Corp. i486 with that of the Gmicro/300. Note that the Sparc chip uses one clock cycle for each instruction. Without a memory operand in each instruction, this processor requires a total of at least three execution

cycles without register conflicts or a memory wait cycle.

The i486's five-stage pipeline executes the same operation in one instruction (for example, an Add reg to mem). It is better for external bus traffic and the cache hit ratio that the CPU has a complex addressing modifier because the instruction length is shorter than the CPU cycle with its simple addressing modifier. But, the i486 needs at least three clock cycles to complete the instruction.

The Gmicro/300, with the same number of pipeline stages as the i486, needs only one clock cycle to complete the instruction.[5,6] How can the Gmicro/300 complete the register-to-memory operation in one cycle?

Table 3 compares the execution cycles of a memory-to-memory operation. The Sparc processor takes at least four cycles to complete the operation. The i486 needs two instructions to execute the operation because its instruction format can't define a memory-to-memory operation. So, the i486 takes at least four cycles to complete the operation.

The Gmicro/300, however, needs only one instruction for a memory-to-memory operation. The instruction format of the TRON architecture specification can define both source and destination operands as memory operands in one instruction. As a result, the Gmicro/300 completes the operation in two clock cycles.

How does the Gmicro/300 complete the operation in half the clock cycles of the i486? The pipeline structure of the Gmicro/300 is tuned for memory operation, while the i486 pipeline is not.[7] The i486 pipeline stages are:

1) instruction fetching and alignment,
2) first instruction decoding,
3) second instruction decoding (if needed, the chip calculates a memory address here),
4) operand execution, and
5) operand store.

Here it is unclear whether we should count instruction fetching as one of the pipeline stages. The Gmicro/300 pipeline becomes six stages with the same count rule as the i486. But, instruction fetching continues until the instruction prefetch queue fills. Instruction fetching remains separate from instruction execution except when executing branch instructions. So, it may be proper not to count instruction fetching as one of the pipeline stages.

From this point of view, the i486 pipeline now becomes four stages long. It doesn't require the operand fetching stage that the Gmicro/300 defined as one of its pipeline stages. This is the main reason the Gmicro/300 needs fewer clock cycles than the i486 to execute memory operations. Essentially, memory operations must define operand fetching as one of the pipeline stages, or execution slows.

Table 2.
Comparing a register-to-memory operation.

Processor	Operation steps	Clock cycles
Sparc	Load mem to reg2	1
	Add reg1 to reg2	1
	Store reg2 to mem	1
	Total	3*
i486[6]	Add reg1 to mem	3**
Gmicro/300	Add reg to mem	1†

* Without register conflicts.
** Needs three cycles to complete one step.
† Needs one cycle to complete one step.

Table 3.
Comparing a memory-to-memory operation.

Processor	Operation steps	Clock cycles
Sparc	Load mem to reg1	1
	Load mem to reg2	1
	Add reg1 to reg2	1
	Store reg2 to mem	1
	Total	4*
i486[6]	Mov mem to reg1	1
	Add reg1 to mem	3
	Total	4
Gmicro/300	Add mem to mem	2
	Total	2

* Without register conflicts.

When a processor uses a memory operand, the address translation mechanism incurs overhead. Pipeline designers must decide whether to assign one stage for the address translation: The more pipeline stages, the bigger the performance degradation brought by branch instructions. Degradation occurs because pipeline processing of instructions after the branch instruction will be cancelled when a branch is taken. It is desirable not to increase the pipeline stages with the address translation mechanism.

One way to solve the problem is to adopt a logical cache memory that can be accessed by a pretranslation address. The Gmicro/300, however, contains physical cache memories that can be accessed only by the trans-

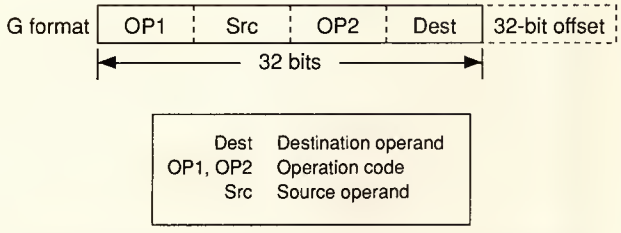


Figure 4. A sample instruction format.

lated address. It must compare an external physical address with the internal cache tags for a bus "snoop" (address detection) function. If the address translation and operand fetching take place in separate stages, the pipeline extends to six stages. At most, four instructions will be cancelled when a branch is taken in a six-stage pipeline.

Let's look at the pipeline structure of the Gmicro/300. We've already stated that memory address calculation and memory operand fetching occur in different stages. The address translation and accesses of the physical cache memory take one clock cycle for the memory operand fetching stage. We adopted a dynamic circuit that acts as a domino logic between the TLBs, each cache's tags, comparators of the TLBs and the tags, and each cache's memory data selectors. We gain some margin of the delay time because the domino logic does not demand clocked latches in midcourse.

The pipeline structure includes:

1) *DC, the instruction decoding stage.* The Gmicro/300 decodes an instruction code such as the G-format instruction with a 32-bit memory address offset field in one clock cycle. Decoding involves six programmable logic arrays, three of which decode operation code and three that decode the addressing modifier.

2) *AC, the memory address calculation stage.* The Gmicro/300 calculates a memory address to fetch or store a memory operand. The TRON architecture specification defines an addressing mode such as base address + index × scale + offset. The processor contains a special adder with three input ports to complete the addressing mode in one clock cycle.

3) *OF, memory operand fetching stage.* The Gmicro/300 executes the address translation and fetches a memory operand. When the memory operand is only written (as in store instructions), the processor checks the write protection.

4) *OE, operation stage.* The Gmicro/300 executes operations.

5) *OW, operand write stage.* The Gmicro/300 writes result operands to memory or to registers. A buffer stores memory operands. This buffer holds the write address, the data to be written, the instruction address that stores the data, and other attributes of the data.

Other one-cycle structures

A 64-bit-wide bus connects the instruction cache, instruction queue, and instruction decoding units. As shown in Figure 4, the bus transfers 64-bit instructions such as G-format instructions with a 32-bit offset as an address modifier in one clock cycle.

The execution block consists of two left-right, two-way barrel shifters; a mask pattern generator; and a logical unit that differs from the ALU. The shifters operate in one cycle. The mask pattern generator and the logical unit manipulate bits in one cycle. The ALU provides sign extension circuits in front of input ports that offer one-cycle sign extension and zero extension.

The two independent, internal cache memories contain two TLBs that are also independent. An instruction fetch and an operand access execute in one cycle. Later, we show the qualitative effect of the separate caches.

Each 2-Kbyte internal cache is a two-way set-associative memory and a physical cache that includes physical addresses in tags. The Gmicro/300 does not restrict operand alignment. A special-purpose selector in the internal, write-through operand cache accesses nonword-aligned data to complete the access in one clock cycle.

Effects of internal caches

Many microprocessors now provide internal cache memories, typically using two methods of implementation. One CPU may contain a unified internal cache for the instructions and operands (like the i486). Another may support two separate internal caches for instructions and operands (Intel's i860). The Gmicro/300, however, tries to complete the memory-to-register operation in one clock cycle by executing an operand access of a current instruction and fetching a following instruction simultaneously. So, we adopted two separate cache memories in the Gmicro/300.

The cache memories, however, have their own performance problem:[8] Their efficiency is influenced by cache misses. The cache hit ratio depends on the cache size and a block size that must be replaced. The external memory access time and the total time of data transfer to fill up a block of cache memory determine the wait time: The higher the hit ratio and the shorter the wait, the higher the system performance.

The instruction cache and the operand cache hold 2 Kbytes each. Use of a nibble mode of dynamic RAM in an external main memory ensures that the block size is 16 bytes.

As Figure 5 indicates, the Gmicro/300 supports a burst transfer mode to decrease the wait time. We have two burst transfer modes: Fast, which allows a memory wait state only at the first word transfer, and slow, which allows a memory wait state at every word transfer. Here, we consider the fast mode and the nonburst mode.

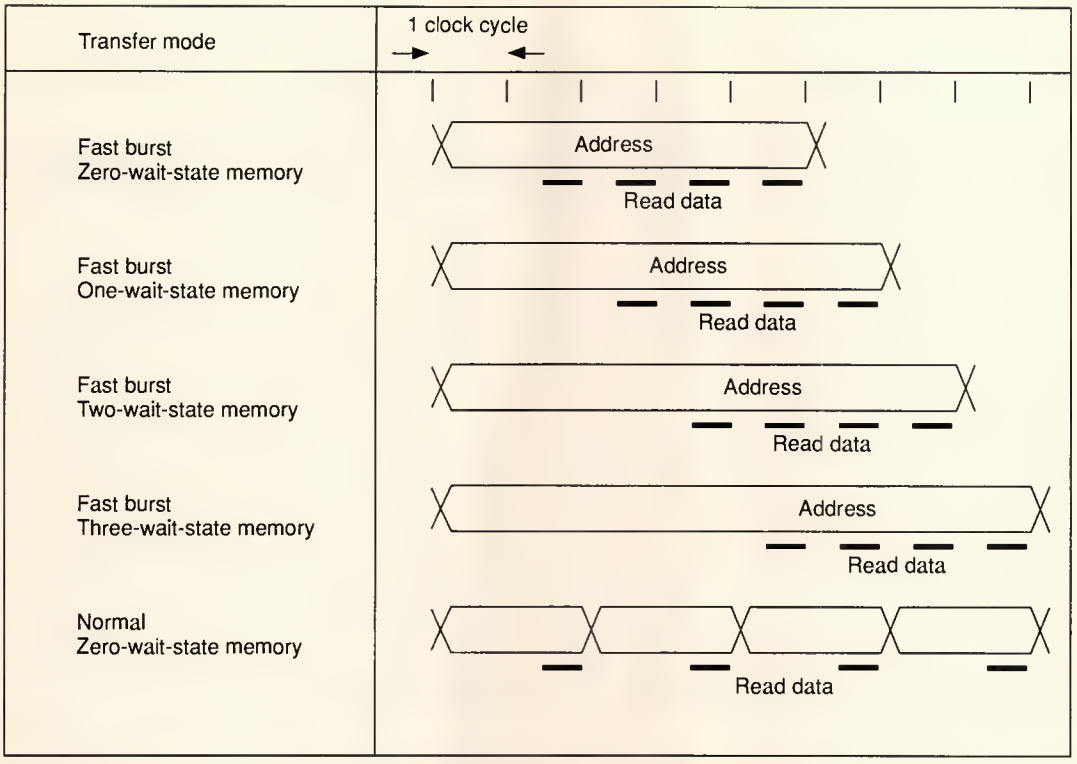


Figure 5. Bus timing in the Gmicro/300.

We measured both modes for total performance, cache hit-ratio, and external bus traffic. We used the Dhrystone Version 1.1 program to measure the parameters. This program was compiled by the GNU C compiler Version 1.36 whose code generator we changed to accommodate the TRON instruction set. The program's static length is 876 bytes, and the operand size fetched by the program is less than 288 bytes. (GNU is publically licensed software copyrighted by Free Software Foundation, Inc. in 1989. We received permission to use it internally; it is not available for commercial purposes.)

After the first loop, all of the instructions move into the internal instruction cache, and all of the operands move into the internal operand cache. Therefore, after the first loop, the external bus activity consists only of write throughs from the write-through cache. Gmicro/300 executes a general program near the first loop, because after the first loop all instructions and all operands are fetched from the internal caches. The loop includes 357 half words, and the total decode length is 472 half words. The instruction length averages 2.64 bytes per instruction. We show the results in Table 4.

Effects of internal caches. Table 4 compares the total required clock cycles listed under condition 1 and condition 2 (first loop) to show the efficiency of the internal caches. When the cache is turned off, Gmicro/300 takes 1,621 cycles, and when the cache is on, 1,051 cycles. The internal caches improve total performance by up to 35 percent. The instruction cache's hit ratio is 63.1 percent at the first loop. The Gmicro/300 executes a general program near this hit ratio.

Effects of burst transfers. We can determine the efficiency of the burst transfer mode by comparing the total clock cycles listed under condition 2 (first loop) and condition 3 (first loop). We assume zero-wait-state memory. Condition 2, the nonburst mode, needs four separate bus cycles (eight machine cycles) to fill up one block of the internal cache. Condition 3, the fast burst mode, needs five machine cycles to fill up one block. We experienced a total performance improvement of 14.8 percent in the burst transfer mode.

Effects of memory wait states in burst transfer mode. As Figure 5 shows, one memory wait state in the fast burst transfer mode delays the availability of a word needed by the current instruction. In Table 4 we can see the effects of memory wait states in the burst transfer mode with the comparison of the total clock cycles from condition 3 to condition 5. Total performance degrades 12.5 to 13 percent with each additional wait state in the memory access.

Table 4.
Comparing total clock cycles using the Dhrystone program, Version 1.1, and the GNU C compiler, Version 1.36. Average instruction length is 2.64 bytes.

Condition	1	2		3		4		5		6	
CACHE	OFF	ON		ON		ON		ON		ON	
Transfer Mode	Normal	Normal		FAST burst		FAST burst		FAST burst		FAST burst	
Memory wait	0	0		0		1		2		3	
Loop	1'st	1'st	2'nd	1'st	2'nd	1'st	2'nd	1'st	2'nd	1'st	2'nd
(1) Total clock	1621	1051	710	895	710	1011	746	1142	788	1285	855
(2) Operand access count	262	262	262	263	262	263	262	263	262	263	262
(3) Operand write count	105	105	105	105	105	105	105	105	105	105	105
(4) Operand cache miss count	262	77	11	78	11	78	11	78	11	78	11
(5) Operand read bus clock	318	144	0	90	0	108	0	126	0	144	0
(6) Instruction access count	209	244	297	257	297	253	297	243	297	242	297
(7) Instruction cache hit count	0	154	296	169	296	166	296	155	296	154	296
(8) Bus busy clock	1220	768	210	557	210	732	315	907	420	1074	525
(9) 1-cache,0-cache simultaneous access count	—	82	98	69	98	62	99	55	97	57	87

Effects of store buffer. The Gmicro/300 contains one store buffer. With multiple store buffers, it takes a long time to execute an interrupt acknowledgment due to two factors. 1) The CPU can't execute the acknowledgment until the store buffers empty, and 2) the CPU's exceptional operation, such as external bus errors, becomes very difficult.

In Table 4, note that all second-loop external bus clock cycles are store cycles. We can discover one effect of the store buffer by changing memory wait states and measuring total clock cycles from condition 3 to condition 6. Accordingly, without a store buffer, the total clock cycles would increase by a part of external bus clock's increase. From zero to one wait states, we can note an improvement of 8.5 percent; from zero to two, a 14.3-percent improvement; and from zero to three, a 16.5-percent improvement. These values depend on how often stores are performed, but they show us that a store buffer is more useful with a slower memory.

Effects of internal cache separation. Cache separation is an important parameter in an internal cache architecture. We measured the effects of cache separation by using a logic simulator. Row 9 of Table 4 shows us a count of simultaneous cache accesses to an instruction cache and an operand cache. If the caches are not separate, the total clock cycles will increase by the count of the simultaneous cache accesses. Under a nonburst transfer mode in the first loop, Gmicro/300 took 82 clock cycles, which is 7.8 percent of the total cycles. In the second loop with zero-wait-state memory, Gmicro/300 took 98 clock cycles, or 13.8 percent of the total cycles.

Despite the use of relatively small internal caches, the Gmicro/300 microprocessor achieves high performance by combining several features. It combines separate caches, a burst transfer mode, a five-stage pipeline along with an explicit definition of the operand fetching stage in the pipeline, one-clock-cycle address translation and cache access, and one-clock-

cycle execution of a memory-to-register operation. Currently, we seek even higher performance in the Gmicro/300 by increasing the clock rate and including a floating-point operation unit in the CPU.

Acknowledgments

We thank Ken Sakamura, professor of the University of Tokyo, and two members of the Gmicro group, Hitachi Ltd. and Mitsubishi Electric Corp., for their cooperation.

References

1. K. Sakamura, "Architecture of the TRON VLSI CPU," *IEEE Micro,* Vol. 7, No. 2, Apr. 1987, pp. 17-31.
2. K. Sakamura, "TRON VLSI CPU: Concepts and Architecture," *TRON Project 1987,* Springer-Verlag, Tokyo, 1987, pp. 200-308.
3. T. Kitahara et al., "High Performance Bus Interface of Gmicro/300," *TRON Project 1988,* Springer-Verlag, 1988, pp. 317-329.
4. M. Itoh, "Architecture Characteristics of Gmicro/300," *TRON Project 1987,* Springer-Verlag, pp. 273-280.
5. "The i486 Microprocessor," Intel Corporation, Santa Clara, Calif., Apr. 1989.
6. "The Intel 80486 Strikes Back," *Microprocessor Report,* Vol. 3, No. 4, Micro Design Resources Inc., Palo Alto, Calif., Apr. 1989.
7. J. H. Crawford, "The i486 CPU: Executing Instructions in One Clock Cycle," *IEEE Micro,* Vol. 10, No. 1, Feb. 1990, pp. 27-36.
8. A.J. Smith, "Cache Memories," *Computing Survey,* Vol. 14, No. 3, Sept. 1982, pp. 473-530.

Takeshi Kitahara is a senior engineer in the Gmicro CPU Design Group, Microcomputer Development Department of Fujitsu Limited. He helped develop a communication system between a host computer and a digital facsimile. He has also been engaged in the development of the Gmicro CPU and the architecture design and logic design method of full-custom LSI chips.

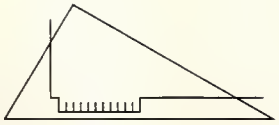
Kitahara received the BE degree in applied physics from the University of Tokyo. He is a member of the Information Processing Society of Japan.

Taizo Satoh is an engineer in the same Gmicro CPU Design Group. He has helped develop the Gmicro CPU and pipeline architecture as well as the cache control method of the 32-bit microcomputer.

Satoh received his BE degree in electrical engineering from the Osaka City University.

Questions concerning this article can be addressed to Takeshi Kitahara, Fujitsu Limited, Microcomputer Development Dept., 1015, Kamikodanaka Nakahara-ku, Kawasaki 211, Japan.

Department



On the Edge

Carl Warren
McDonnell Douglas Space Systems
(714) 896-3311, ext. 6-0669
MCI Mail: 310-9380; Compmail+ c.warren

Realizing a transmission model

A reader recently informed me that I had led many a reader down the proverbial technology garden path in the February issue. The statements at issue involved frequency and impedance.

I said that frequency could be related to the movement of the electronics in a conductor. I also said that impedance was the derivative of frequency. However, impedance is not derived from frequency. It is a function of frequency and "transforms" current to voltage. You can think of impedance in many ways, but it really is a transfer function and an analogy to Ohm's law. My goal with the discussion was to communicate the notion that you can visualize the process if you think of it at the smallest level.

Before you can "realize" a model of a transmission system, you must have some conceptual idea of what is happening physically. This process is similar to mentally folding a box, hoping you make the right choices. I don't want to belabor the point I was trying to make. Possibly I'm guilty of oversimplification; I'll try to avoid it in the future.

So that we can proceed with the discussion on SPICE modeling (I'll explain that later), let's review some more of the basics.

A lumped world

The resistance in a piece of wire is distributed throughout its length. This idea is not so difficult to grasp when you think about it. However, it's often easier to consider the resistance as being concentrated into one lump. You can think of impedance in the same manner.

Of course, when you lump something, some commonality must exist between the two or more sources of the combined resistance or capacitance. For circuits, assume they have the same current and voltage for each branch. If they differ at any branch, you can't lump them. This difference, of course, results from whether the current is series or parallel.

Suppose you have a network of inductances and resistances, with each branch carrying a different current. You end up with different voltages flowing across the terminals. At McDonnell Douglas, we deal with this issue when considering the various remote terminals on a Mil-Std 1553B transmission line (see figure). We know that as the length varies, we end up with some changes in current, inductance, resistance—and hence voltage—at the terminal.

In earlier columns, I talked about transmission lines as having some wavelength. The wavelength relates to the frequency of the current coursing though it. The variance in frequency influences the way you treat the line.

For our discussion, we have to consider that we are dealing with frequencies in the radio-frequency range at 1-MHz rates.

The mathematics of the transmission line tells you how the line works (see the table on p. 78). The computations that you see in the table are part of a model. J.A. Hubbard, a McDonnell Douglas engineer, developed the model using Microsoft's Excel spreadsheet program.

Those differentials

Let's go back to what started this discussion—my mention of how frequency and impedance are related. Some fundamental differential equations tell us a great deal about the line. For example, when you have a line of X length, it's safe to assume that the voltage at the re-

0272-1732/90/0600-0076$01.00 © 1990 IEEE

ceiving end differs from that at the sending end. A change, or delta voltage, exists:

$$\delta V = V_{out} - V_{in}$$

Now you can take this differential and divide it by the length of the section of line *dX*. So you end up with

$$dV/dX$$

This equation pretty much tells you the average drop-per-unit length of line in the desired section. Describing this as *dV/dX* establishes the rate of change of voltage with respect to distance.

This is all pretty simple stuff. Now look at the equations in the table again. Note that they appear to be more complex. In reality none is more complex than the simple formula we just created. Basically what happens is that the electrical characteristics of the transmission line change with respect to the frequency and other parameters. When the frequency becomes either slower or faster, the components of wavelength, impedance, and fields change with it.

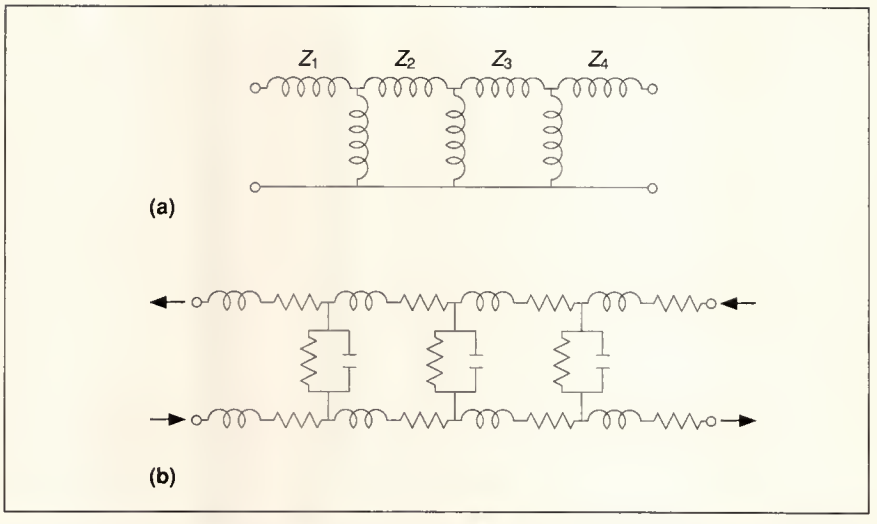
Now let's look at impedance. The input impedance to a transmission line is the ratio of the voltage at the sending end of the line to the current entering the line at the sending end. The length of the line affects the impedance, as does the frequency.

Of course none of this makes any sense at all if you don't understand—or at least appreciate—the characteristics of impedance. Impedance has both resistive and reactive parts. The equivalent resistance accounts for power dissipation in the transmission line. The reactive part of the impedance accounts for phase shift, among other things. The impedance, or Z_0, can be expressed as

$$Z_0 = V/I$$

where *I* equals current. You can see by inspection and using your intuitive knowledge that delta changes in voltage and current can produce a different *Z* at any instant. We can deduce that an open circuit *Z* that is the characteristic impedance *Z* equals infinity yields:

$$Z = -\mathbf{jZ_0}\ \text{Cot}\ \beta x\ (\mathbf{j\beta x = jG}),$$



Skilling's notion of a lumped model network: an example that cannot be easily lumped (a) and an example that can (b), where Z_1 etc. = lumped impedance.[1]

or Euler's formula. Although the formula is general, it gives us a tool to look at a transmission line as a function of frequency, length, and capacitance.

Essentially, you can think of the way the transmission line changes in terms of the hyperbolic functions cosh and sinh. You can stop the rotation at any point and make some general statements.

For example, you can define a line at any frequency. At this frequency it will have an associated *Z*, or characteristic impedance. If you want to treat this line as purely resistive, you can terminate the line with a resistance that is equal to the real or resistive part of the defined *Z*. However, should the frequency or length change, the *Z* changes as well. At a single frequency or for a narrow-band signal, the transmission line can be reasonably matched. For wide-band signals, the matching of a transmission line is extremely difficult.

Understanding the transmission line and how it operates takes more than reading a column or two. Although hundreds of texts exist, I recommend that you pick up a copy of Skilling's *Electric Transmission Lines*.[1] This book is considered the last word in transmission theory. You will want to read it cover to cover and study it until it falls apart—I'm on my sixth copy.

Spicing it up

Once you have satisfied yourself that you are at least comfortable with the nomenclature of a transmission line, you can consider modeling one. When I was a boy in the 50s, just about everyone I knew was a ham, or amateur radio operator. Our goal was to build the ultimate transmitter or receiver or the best antenna on the block.

I labored every weekend, bolting aluminum rods together or clipping twin-lead wire to match specific frequencies. I wanted to get the best ratio of waves (current) on my line. My models in every case were real. When your allowance is $2 a week, the models sometimes become too real, necessitating dipping into hard-earned box-boy money.

Frequently, I missed the boat; I cut off too much wire. Of course, there was the notion of adding on, or "stubbing," to get a match. I learned the ins and outs of small antennas, ground planes, and field effects.

I found I could cut my losses by learning how to use the tools of transmission lines. The standing wave meter was my closest ally—I always kept Heath Kit and Allied Radio catalogs close at hand.

I did make use of the *Radio Amateurs Handbook* and learned how to calculate a line. I also discovered that you could

A model of an unloaded Mil-Std 1553B bus. (Courtesy of J.A. Hubbard.)

Parameters	Symbol	Value(ft)	Units	Value(meters)	Units	Equations
Characteristic Impedence	Zo =	70.00	ohms			
Time Delay (per unit length)	Td =	2.212	nSec/ft	7.26	nSec/m	Zo*C/ (unit length)
Input Voltage to Bus	VI =	9.00	volts			
Line Loss (specification)	dB =	-1.50	dB/100 ft			
Resistance (ac)	R =	0.099	ohms/ft	3.25E-01	ohms/m	R(ac)= R(dc)(1/(2(sigma/r)-(sigma/r)^2)
Conductance	G =	3.28E-16	mhos ft	1.00E-16	mhos m	
Length	Len=					
Frequency	F =	1.00	mHz			
Distance Between Wires	d =	0.002538889	f t	0.000064	m	
Radius of Wire	r =	0.001233333	f t	0.000031	m	
Resonent Frequency	Rf =		mHz			1/SQRT((Lo*Len)*(Co*Len))
Angular Frequency	ω =	6.28	mHz			2*PI*F
Voltage Out	Vo =		volts			(VI*10^(dB*((Len/.3048)/100)/20))
Capacitance (lumped)/unit length	Co =	31.60	pF/ft	1.037E-10	f/m	From measurements
Line inductance/unit length	Lo =	154.84	nH/ft	5.079E-07	H/m	From Zo=((R + jwLo)/(G+jwCo))
Current	Io =	1.29E-01	amps			VI/Zo
Total Time Delay	Γ =		nSec			(Len*SQRT(L*C))
Wavelength	λ =	452.14	f t	137.81	m	
Velocity	v =	4.52E+08	ft/sec	1.38E+08	m/Sec	1/SQRT((Lo)*(Co))
Length of Stub	Ls =	1.00	f t			
Capacitence of Connector	Cd =	0.50	pF			Co*(2/12)
Termination Resistance	Zr =	70.00	ohms			
Capacitance of Stub	Cs =	31.60	pF			Cd*Ls
Char. Imped. with R,G,C,L, real	Zo(real) =	70.08	ohms			((R+jwL)/(G+jwC))^0.5
Char. Imped. with R,G,C,L Imaginary	Zo(Imaginery) =	-3.56	ohms			
Magnitude Char. Impedance	Zo =	70.17	ohms			((R+jwL)/(G+jwC))^0.5
Unloaded Impedence	Zu =	69.62	ohms			Zo/SQRT(1+(Cd/Co))
Reflection Coeff. from termination, real	Γ =	0.00066	real			((Zr/Zo) -1)/((Zr/Zo)+1)
Reflection Coeff. from termination, Imag	Γ =	-0.00103	Imaginary			
Impedance of Stub	Zs =	70.00	ohms			
Line Input Impedence	ZI =		ohms			Zo*(ZrCoshGl + ZoSinhGl)/(ZoCoshGl + ZrSinhGl)

Values of the 1553 bus varying over length										Values of 1553 bus varying over freq.	
Length (meters)	Total Capacitence pF	Total Inductance nH	Γ nSec	Resonant Freq. (Rf)	ZI Mag.	V Out (dB)	V Out (Computed)	dB/100ft	% Attenuation	Frequency (mHz)	Zo(w) Mag.
0	0.00E+00	0.00E+00	0.00E+00	0.00E+00	69.62	9.00	9.00	0.00	0.00	0.0	5.70E+07
5	5.18E+02	2.54E+03	1.15E+12	8.72E-04	69.81	8.75	8.90	-0.61	1.15	0.1	83.61
10	1.04E+03	5.08E+03	2.29E+12	4.36E-04	69.75	8.50	8.79	-0.61	2.29	0.2	74.14
15	1.56E+03	7.62E+03	3.44E+12	2.91E-04	69.78	8.27	8.69	-0.61	3.42	0.3	71.92
20	2.07E+03	1.02E+04	4.59E+12	2.18E-04	69.90	8.04	8.59	-0.61	4.53	0.4	71.10
25	2.59E+03	1.27E+04	5.74E+12	1.74E-04	69.68	7.81	8.49	-0.61	5.63	0.5	70.71
30	3.11E+03	1.52E+04	6.88E+12	1.45E-04	69.99	7.59	8.40	-0.61	6.72	0.6	70.49
35	3.63E+03	1.78E+04	8.03E+12	1.25E-04	70.00	7.38	8.30	-0.61	7.79	0.7	70.36
40	4.15E+03	2.03E+04	9.18E+12	1.09E-04	69.96	7.17	8.20	-0.61	8.85	0.8	70.27
45	4.67E+03	2.29E+04	1.03E+13	9.68E-05	69.93	6.97	8.11	-0.61	9.90	0.9	70.21

more or less create the line on paper and envision its performance without buying wire.

Apparently, large companies had made some similar discoveries. The result was the creation of modeling or simulation tools. One of the more popular is SPICE (Simulation Program with Integrated Circuit Emphasis). This powerful tool is very useful in visualizing a circuit. Of course other good tools exist, but I've limited the discussion to SPICE.

Depending on how you look at SPICE, you can say that it doesn't do a good job at transmission lines, which is basically true. Then you pick up your copy of Skilling and realize that you can make use of the lumped model technique. You do have to forget about such things as mutual coupling, cross talk, and ground plane effects. You can manage these phenomena, but they are indirectly related to the SPICE model.

If you assume an ideal line (one with no loss), the simulation is straightforward. In fact, you can use the same approach Hubbard did with his Excel model. The results are ideal for a given domain and good enough for most applications. What you want to see is the "lossy" effect.

This effect is due to a number of frequency-dependent effects that cause losses of signal over a line. You can model these lines using time-domain analysis.

In our models of Mil-Std 1553B we look at a multiconductor transmission line. We have to contend with coupling as well as changing capacitances, inductances, and varying impedances. The line current isn't necessarily constant from remote terminal to remote terminal. (However, the zero crossing value must remain above 1 volt for a period of 50 nanoseconds or better to get the phase-locked loop to respond.) This requirement is very specific to our analysis and shouldn't be considered a standard for all situations.

The model is just as complex as it appears. You can, however, rely on the intuitive knowledge of Skilling. Treat each element of the transmission line as a separate lumped circuit and analyze it separately. Then use this information to describe the final transmission line as one lumped model.

The result is a reasonably effective approximation of the actual 1553B transmission line.

Over the past several months, I have used the 1553B as a model. This transmission line is ideally suited for describing the function of a bus. It is nothing more than a piece of wire. But when this wire is treated as an interconnection device, it reacts in very specific ways.

Unfortunately, I don't have enough time or space to detail SPICE modeling. I recommend that you obtain copies of certain articles.[2-5] You might also want to look at the IEEE Press and Computer Society Press book lists for various books and videotapes on the subject.

Final note

If you have developed a useful transmission line tool, or have created a SPICE model, let fellow Computer Society members know. Send your work to me at the address listed at the beginning of this column.

References

1. H.H. Skilling, *Electric Transmission Lines: Distributed Constants, Theory and Applications*, McGraw-Hill, Hightstown, N.J., 1951, p. 2.
2. A.R. Djordjevice and T.K. Sarkar, "Analysis of Time Response of Lossy Multiconductor Transmission Line Networks," *Proc. IEEE*, Vol. 75, No. 6, June 1987, pp. 743-764.
3. C. Hymowitz, "Modeling Interconnects in SPICE," *RF Design*, Vol. 10, No. 1, Jan. 1990, pp. 49-54.
4. C.R. Paul, "A Simple SPICE Model For Coupled Transmission Lines," *Proc. 1988 IEEE Symp. Electromagnetic Compatibility*, IEEE Press, Piscataway, N.J., 1988, pp. 327-333.
5. C.S. Yen, S. Fazarinc, and R.L. Wheeler, "Time-Domain Skin-Effect Model for Transient Analysis of Lossy Transmission Lines, *Proc. IEEE*, Vol. 70, No. 7, July 1982, pp. 176-181.

Micro News *(continued from p. 8)*

Neural net progress

Stanford University scientists recently installed a "nerve chip" into animals in an attempt to develop electronic control of human artificial limbs. Although human implants are about 10 years away, a 1 × 3.6-mm silicon chip has survived for a year between the severed ends of a foot nerve in a rat. Individual axons of 2 to 6 μm in diameter regenerated from the ends of the severed nerve through 8 to 16-μm-wide holes at the end of the chip and then reconnected.

Project leaders hope to create an interface between the nerves in an amputation stump and a robot limb. Bernard Widrow is working on a neural network that would reside in the artificial limb and respond to the electrical patterns coming from the patient's brain.

DARPA chose Nestor Inc. and Intel Corp. to codevelop a neural network chip that can process 150 billion synapse interconnections per second. Plans call for a 1,000-neuron device that becomes more precise and intelligent as it learns.

Researchers predict that neural networks based on this chip will perform 10,000 times faster than present software systems. Intel's proprietary Flash memory technology used in 250,000 memory cells will provide nonvolatile storage for synaptic weights, neuron references, and control of classification outputs.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 183 **Medium** 184 **High** 185

Micro Standards

Carl Warren
McDonnell Douglas Space Systems
(714) 896-3311, ext. 6-0669
MCI Mail: 310-9380; Compmail+ c.warren

The Scalable Coherent Interface

IEEE P1596 (the Scalable Coherent Interface) promises to become one of the more important standards of the 1990s. To this end, I asked the working group—chaired by David Gustavson—to give us the following glimpse of the proposed standard.

The continually increasing demand for more processing power apparently has no limit. One can saturate the resources of any computer trivially by merely specifying a finer mesh or higher resolution for the solution of some physical problem like hydrodynamics. Consequently, engineers and scientists have become desperate for enormously large computers.

It seems necessary to use a large number of processors cooperatively to obtain this kind of computing power. Single-chip processors heavily employ pipelining and similar large-mainframe tricks. Vector processors help, but they are not very efficient in many applications. Multiprocessors that communicate by message passing work well for some applications, but not for all.

The shared-memory multiprocessor looks like the best strategy for the future, but a great deal of work in software development must occur first to aid multiprocessor efficiency.

The working group thinks it is important for the standard to support both the shared-memory and the message-passing models efficiently. It is also important to support optimal software for a wide range of problems. These conditions are especially necessary for a system that dynamically allocates processors and perhaps changes its configuration depending on the nature of its load.

With these important factors in mind, we started SCI to meet the needs of new generations of processor chips—some of which can saturate the fastest buses single-handedly. We attempted to increase the bandwidth of a backplane bus past the limits set by backplane physics. Experimentation indicated that we had to abandon the bus structure to achieve our goals.

Physics (distributed capacitances and the speed of light) and the one-at-a-time nature of a bus limit backplane performance. To gain performance far beyond that of buses and backplanes, we need better signaling techniques. Rather than using bused backplane wires, SCI employs point-to-point interconnection technology. This design approach eliminates many of the physics problems and results in much higher speeds.

The switch from a shared backplane bus to a point-to-point interconnection created many new problems and research topics that were resolved in record time in this SCI project. However, much research remains to be done to determine optimal ways to use the mechanisms SCI provides. SCI also requires the development of novel arbitration and cache-coherence protocols, which makes the project a challenging one indeed, particularly in view of our time goals.

Historical perspective

Most of the developers of SCI come from high-speed-bus backgrounds, such as Fastbus (IEEE 960) or Futurebus (IEEE 896). Paul Sweazey, who was the coordinator of the Futurebus Cache Coherence task group, initiated a Superbus study group under the IEEE Computer Society's Microprocessor Standards Committee in November 1987. The purpose was to avoid the multitude of competing incompatible standards we saw in the 32-bit generation. The Futurebus group tried to solve that problem in the late 1970s but could not converge in time to head off the development of many alternatives.

The Superbus study group met for less than a year before deciding that there was indeed a way to do better. They saw a method for achieving the throughput rates that are required to support multiple 100-Mflops (million

0272-1732/90/0600-0080$01.00 © 1990 IEEE

floating-point operations per second) processor chips, namely about 1 Gbyte/second per processor. As the Superbus study group became the SCI working group, we were particularly urged on by Paul L. Borrill, Futurebus chair, and John Moussouris (one of the founders of MIPS). Moussouris frightened us all with his predictions of immensely powerful processors in the near future—which already seem to be coming true!

We applied for a project authorization in July 1988, which was approved by the Standards Board in October. David Gustavson was appointed to chair the SCI group, and David V. James became the logical task group's coordinator and vice chair. Gustavson also served as physical task group coordinator, handled the mailings, and shared taking the minutes with James.

We started a Control and Status Register (CSR) and I/O architecture effort within SCI, based on some significant contributions by James. Because we recognized that this effort would be important for other standard buses as well, we split it off as an activity to be shared with Futurebus+, Serial Bus (P1394), and others. In April 1989, this effort also became an official project—P1212—chaired by James.

Moussouris frightened us all with his predictions of immensely powerful processors in the near future.

The establishment of a standard for control and status registers and I/O—attempted many times before—now seems finally within reach. We credit the improved progress to James, who brought considerable architectural experience to bear. He generated sufficient rationale for the various choices so that the decisions no longer seemed entirely arbitrary.

The CSR standard project has become a unifying force for the last of the previous generations of buses (Futurebus+). This development encouraged VMEbus and Multibus-II users to employ the CSR architecture as part of their move to Futurebus+. It also facilitated a future move to SCI as system requirements grow. In this way, we developed a relatively smooth and well-defined growth path from present-generation, single-processor systems through Futurebus+'s few-processor systems with cache coherence, all the way to SCI's many-processor systems.

Because of our priority on interfacing SCI to other buses, we included protocol hooks that would not otherwise be needed. In exchange, SCI users can draw on a large number of existing simple I/O interfaces, which makes getting started much easier.

Other groups added momentum to the SCI effort. March 1989 marked the beginning of the Fiber Optic task group (SCI-FI) led by Hans Wiggers. An SCI/Futurebus+ Bridge task group also began at that point, led by Mark Williams (a joint appointment with Futurebus+).

During the development of the standard, Knut Alnes and Ernst Kristiansen worked on an early implementation of SCI and provided input for the details of the specification. They also initiated work on formal verification of the cache coherence mechanisms.

Another major contributor, David James, generated documents at an incredible rate. The bulk of the June 1989 specification for SCI resulted from his single-handed effort. At the same time he was producing two volumes of similar size for the CSR working group! He believes that having something on paper produces more productive discussions. Our experience seems to support that view.

We felt that we would have to finish SCI in record time to avoid the chaos of the 32-bit-bus world. Normal development time for a bus standard that involves new design without major historical constraints runs about eight years. To counteract this trend, we worked at a feverish pace, with multiday meetings every month and a great deal of work between. Many workers put in a nearly full-time effort.

One benefit of the increased pace is that the progress improves more than proportionally because one does not have time to forget information between meetings. The result is that the work goes faster and has higher quality and coherence, as we hope you will agree when you examine the final specification.

The normal development time for a bus standard that involves new design without major historical constraints runs about eight years.

A standard interface

Although defining a scalable architecture remains the prime interest of the working group, one of the key issues we faced was the notion of a standard interface. With this in mind, we set about to decide what requirements should be met:

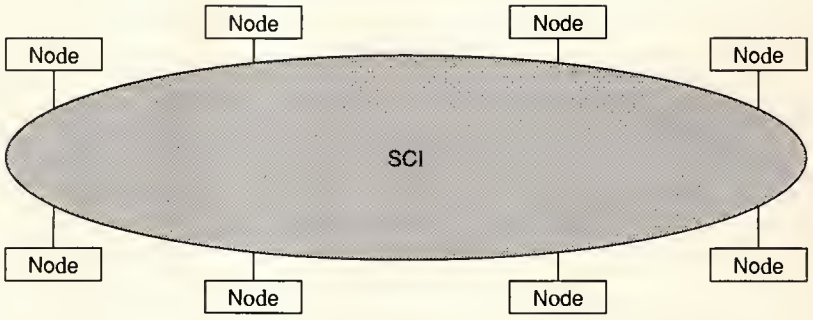
- support a 1-Gbyte/s processor (minimum) with the capability of scaling upward to meet future needs;
- support one to an infinite number of diverse processors; or
- provide for modular growth, thus reducing processor and system cost.

In addition, we realized that it would be necessary to define such things as board size, connectors, power requirements, cooling, electrical signals, and operating protocols.

Multiple processor support

Having more than 100 Mflops on the desktop isn't that far away. The interface system, however, presents a stumbling block. Parallel processors with

shared memory represent the best all-around solution for more computing power. But parallel processors demand more interface bandwidth. One approach is the message-passing system, one that works for some applications but isn't well suited to those that demand maximum bandwidth on a transfer. A 100-Mflops machine demands at least 1 Gbyte/s of bandwidth to avoid system waste and data starvation.



The arbitrary interconnection scheme looks like the familiar communications blob.

Beyond backplanes

Backplane buses like VMEbus, Multibus II, and Futurebus+ have a place in today's system architectures. However, due to the basic physics of a 100-nanosecond limitation, they eventually give up. Furthermore, no matter how robust the arbitration protocol, only one "talker" can exist at a time.

The solution appears simple—at the outset. Learn a lesson from the communications people. Use packets instead of words, handshake with a packet protocol, and employ many unidirectional data paths. Finally, use a series of small rings to provide the interconnections.

Consider the interface

Interestingly, most bus standards are more concerned with how the physical connection occurs rather than the actual interface. SCI is more concerned with the embodiment of the interface, protocol, and the electrical communication mechanism. The unspecified interface used by SCI essentially constitutes the amorphous blob that communications people are accustomed to seeing. It accommodates the nodes that are needed (as seen in the figure).

One of the keystones of P1596 is that it defines scalability. We may or may not know, at any given time, how many processors will be used. For that matter, how many processors will be doing anything useful that needs a bus interconnection? The best we can do at scaling is $N/\text{Log}\ N$. Thus an infinite number of processors can exist.

We need to provide for a large number of processors as well as allow for N number of messages at any instance (thus the requirement for a large bandwidth).

Objects are messages

Because of the current emphasis on object-oriented programming, object-oriented databases and message-passing methods may prove to be the proper approach. The implication, though, is to use shared memory. Designers have debated the notion of shared memory for some time. Having processors share the contents of a central memory is clearly a good idea. A message in memory is available to any processor. This method works well when the processors are closely coupled. Say two processors happen to be on the same board and share the same local bus. Even a single-ported memory provides a "fast" path to the message.

When the processors are separated via the bus, the processors remain the same. Only the latency changes. Some bus interaction must take place. Now performance comes into play. If the message is big, some large amount of time can elapse before any given processor takes advantage of the data. The processor may move the message from shared to local memory, thus improving the locality of the data. Once this is done, the performance related to that Nth processor is improved. However, the overall system may still suffer from degradation.

Now consider N processors that are tied together via a link with infinite bandwidth. Theoretically, the "shareability" of the message is not the case. Indeed, the complexity grows, and a question arises about the plausibility of a many-to-one relationship. Can many processors lay claim to the message at the same time? And if they do, how can they all be sure that the message is the most recent?

These issues are just some of the ones addressed by the SCI group. The proposed standard is moving rapidly. If you have an interest in the detailed standard documents, please contact SCI chair David Gustavson, Stanford Linear Accelerator Center, Bin 88, PO Box 4349, Stanford, CA 94309; phone (415) 926-2863, fax (415) 961-3530; Email DBG@SLACVM,bitnet. If you become active in the working group, you will also be privy to large amounts of technical information on caches, coherency, and fiber-optic interfaces.

In August we will look at P1104.0/D2: A Guide to Backplane Electrical Performance Measurements.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 189 **Medium** 190 **High** 191

Micro Law

Richard H. Stern
1300 19th Street, NW, Suite 400
Washington, DC 20036

Professional ethics and the law

Shown in the box on the next page is a reprint from *IEEE Spectrum* (Apr. 1990, p. 19) of a hypothetical case study. *Spectrum* raises the question of whether the engineers working for the consulting firm and the city official behaved "ethically."

That may be an interesting question, but it is essentially unanswerable. No generally accepted system of primitive terms, axioms, postulates, inference rules, or whatever, exists for deciding what behavior by electrical engineers and customers is ethical. Certainly, whatever ethical system the National Society of Professional Engineers (NSPE) uses is neither agreed to nor binding on most engineers. That includes engineers working for consulting firms or the firms' prospective customers, whether they are city officials or ordinary businessmen.

I propose to address several different questions, therefore, which the *Spectrum* case study suggests. The first is whether the engineers and client behaved legally—that is, would the consulting firm succeed in a lawsuit against either? Second, what, if anything, could the consulting firm do about it?

Essentially, the NSPE response on ethics parallels the law. As a general matter, the consulting firm cannot prevent the client from doing business with whomever the client wants—absent a contract, confidential relationship, or legally recognized proprietary right preventing it. Similarly, while the employees owe a certain amount of loyalty to their employer, they probably are not legally prohibited from quitting their jobs to set up their own consulting firm should a client approach them with an offer. This is true even if the client asks them to work on a project that the employer would prefer to gain for itself. Again, we must presuppose that no contract, trade secret, or proprietary right stands in the way.

To be sure, employers frequently sue employees who quit and go into business in competition with the employer. Frequently, the time, diversion of energy, and cost of litigation puts the employees' start-up out of business.

But this consequence does not necessarily come about because the employees in fact violated a legally protected right of the employer. It often results from a glitch in our legal system, which does not redress all wrongs done to everyone. In some instances, abusive suits may make the ex-employer liable for damages under the antitrust laws, unfair competition laws, doctrines against tortious interference with prospective business advantage, and the like. But having a "cause of action" (a claim on which relief might be recovered against the other party) is not the same thing, of course, as recovering. In any case, my point here is only that the ethical judgment made by the NSPE is probably in accordance with a correct legal judgment, at least in the abstract.

The next question concerns what a consulting firm can do about this undesirable situation, without bringing an unmeritorious lawsuit simply to crush the ex-employees and drive them out of business by bankrupting them. I would suggest several things. First, the consulting firm should have a policy of using written employment contracts. One provision in the contract could be a noncompetition agreement. For example, the employee might agree not to compete with the employer, for one year, for business in the state (or other appropriate area) in which the employer conducts business. In most states, agreements of this type will be upheld and enforced against ex-employees. Further, a provision against ever working competitively on any specific contract proposal that the firm's employee had previously worked on probably would also be upheld.

If the consulting firm and electrical engineers of *Spectrum*'s hypothetical case study had entered into such a contract, the firm would probably be able to obtain an injunction against the ex-employees. The exployees in that event would be prevented from developing the communications system for the city. Moreover, by bringing the contract to the attention of the city, the firm would be able to require it to stop conspiring with or inducing the ex-employees to breach their contract not to compete.

0272-1732/90/0600-0083$01.00 © 1990 IEEE

Ethical judgment

Consider the following case study. It is based on a hypothesis of the National Society of Professional Engineers' (NSPE) Board of Ethical Review. We have made it more germane to our profession through identifying the principals as electrical engineers.

A municipality issues a request for proposals for a citywide emergency communications system. In response, an engineering department head in a large consulting firm assigns the proposal to three of his staff engineers. Soon after the proposal is submitted, the city learns who actually developed the designs, and a city official approaches the three engineers, hoping to contract with them directly, independent of the consulting firm. The engineers inform their department head of these overtures, resign from the firm, and enter into negotiations with the city.

The NSPE ethics board poses the question: was it unethical for the three engineers to take the action they did?

No, it was okay, the board concludes, referring to the NSPE Code of Ethics, which states that the public interest should be put ahead of the interests of the engineers or the consulting firm. The client—in this case, the city—should be able to select the engineer(s) of its choosing.

One solution that the city could have required as a condition of its acceptance of the bid, that the three engineers be assigned to the project, is not mentioned. However, the board implies that the decision might be different if the three engineers gained any "particular and specialized knowledge" while working for the consulting firm. But such information, the board says, must "approach being proprietary in nature."

No mention is made of the implications of the three engineers' development of the proposal on "company time," using the consulting firm's facilities. Also, no ethical problem is apparently seen in the consulting firm's loss of the opportunity to bid on the contract in question, or in how the consulting firm would have profited from assigning the engineers, during the time they spent developing the proposal, to another, ongoing project.

The NSPE board begs the question of whether the city official was unethical in approaching the engineers directly. Nor is the likelihood of the consulting firm bidding on the next project to come along from the city discussed.

—IEEE Spectrum

A second step would be for the firm to place a copyright notice on the proposal that it submits to the city—for example, "Copyright 1990 by John Doe Engineering Corp." Maybe adding "All rights reserved" would be a good idea too. Actually, this addition simply means that the copyright owner asserts its legal rights in South America as well as in the United States, Japan, and Europe. But as a practical matter the phrase may make a reader take the notice more seriously.

This notice will not prohibit the city from using the ideas contained in the proposal. However, the city must not copy the proposal—such as by photocopying it. And it definitely cannot copy any original drawings, whether by photocopying or any other means (such as resketching a copy). If it does, it will be liable for copyright infringement. For example, in a case involving radio station WPOW, the US district court in the District of Columbia held an applicant to the FCC for a station license liable for copyright infringement, because it copied the antenna drawings of an earlier applicant. Part of the relief accorded required the copyist/applicant to withdraw its application from the FCC.

Another possible step would be to try to create a confidential and/or contractual relationship between the client (the city) and the consulting firm. For example, the firm might transmit its proposal with the following language in a cover letter (or include the language into the proposal):

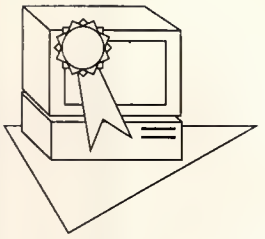
> You will appreciate that we have gone to considerable effort and expense to prepare this proposal for you, and we believe that it reflects original work on our part not commonly known and available to others. We therefore are transmitting this information to you on the understanding that you will keep it confidential, and will not disclose it to a third party or use it without our prior written consent. By the same token, your acceptance and retention of this material will indicate your acceptance of our submitting the information to you on this basis.

Suppose that the city does not respond by returning the proposal with a letter stating, "We cannot accept your proposal subject to the conditions stated in your letter." I believe it highly likely that a court would hold that the firm made an offer to the city to enter into what is called a "unilateral contract." By accepting and retaining the proposal, instead of sending the proposal back, the city agreed to the terms and conditions of the cover letter.

Hence, by subsequently using the information without consent or by disclosing the proposal to another bidder (that is, engaging in "bid shopping"), the city commits breach of contract. It is not clear that a court would enjoin the city from proceeding to let the bid to the ex-employees. But the court probably would hold the city liable for damages—such as the consulting firm's lost profit on the prospective job. Possibly, the city would be enjoined on the theory that it is too difficult to calculate what the damages are.

The NSPE probably concludes rightly that ethical theories will not do much for the employer/consultant in this case study. But some homework on contract writing and some other legal fancy footwork might work wonders.

Department



New Products

Marlin H. Mickle
University of Pittsburgh

Send announcements of new microcomputer and microprocessor products to Managing Editor, IEEE Micro, *PO Box 3014, Los Alamitos, CA 90720-1264.*

Processors and coprocessors

Multiprocessors offer 1.2 Gflops

Sixteen MC860 Series single-slot attached processors link together to provide from 80-Mflops to 1.2-Gflops performance. Based on single and multiple 40-MHz i860 RISCs, the board-level products support computation-intensive applications such as diagnostic medical imaging, radar and sonar processing, simulation and training, and seismic analysis. One model, the MC860VS, provides four i860s on a VMEbus 9U form-factor card.

According to the company, its real-time, multitasking MC/OS operating system is symmetric with the host operating system. In addition, a 40-Mbytes/s bus interface includes a peripheral bus for 80-Mbytes/s direct input and output between the i860 and external devices.

The MC860 products feature C, Fortran, or assembly-language programming capabilities as well as a variety of software tools, a scientific algorithm library, and an optimized scalar math library. **Mercury Computer Systems; from $11,900 (2-Mbyte system).**

Reader Service Number 11

Military chip set operates at 33 MHz

Fully certified for strategic and tactical military use, a 32-bit microprocessor and math coprocessor make use of a pipeline architecture to speed instruction execution. The Military 68030/68882 set conforms to the ANSI/IEEE floating-point math standards, executing a full set of trigonometric and transcendental functions.

Both 33-MHz devices operate in temperatures from – 55 to + 125 degrees centigrade and are packaged as 132-terminal CLCCs or 128-lead PGAs. **Motorola Semiconductor; coprocessor from $2,075 (100s); delivery 8-10 weeks ARO.**

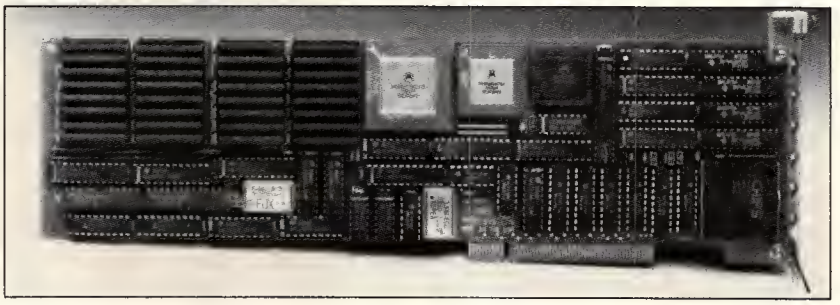
Reader Service Number 12

Speed up Macs with 50-MHz cards

The PowerCard 030 upgrades the Macintosh II, IIx, and IIcx to 25-, 33-, 40-, or 50-MHz speeds compatible with standard SIMM memory. Based on the 68030 CPU with zero-wait-state performance, the accelerators include same-speed 68882 math coprocessors. **DayStar Digital; from $1,495 (25 MHz).**

Reader Service Number 14

Four PS/2 coprocessors operate simultaneously



Yarc Systems Corporation's micro785+ coprocessor system brings to the PS/2 the linear address space, enhanced tools, and floating-point performance of Motorola's 68020 CISC family.

The micro785+ coprocessor system for IBM PS/2 Micro Channel applications performs at 5 VAX MIPS and 3.1 Mflops and provides 4 Mbytes of memory. Up to four coprocessors can operate simultaneously in one PS/2. Programs running on a PS/2 can directly access memory space in 68020 microprocessors. **Yarc Systems; from $2,795.**

Reader Service Number 13

DSP boards

DSP32C processes vectors at 25 Mflops

The full-size, 32-bit floating-point Vector 32C/8500 supports vector processing on IBM PCs, XTs, ATs, and compatibles. Based on the AT&T 50-MHz DSP32C chip, the specialty board comes with a variety of memory options, including a model with no external SRAM at the lowest cost. Other models provide 32 Kbytes to 512 Kbytes of memory; an additional 8 Mbytes of DRAM may be specified on a standard, full-size PC board. **C&C Technology; $4,995 each (512-Kbyte SRAM); OEM discounts.**

Reader Service Number 15

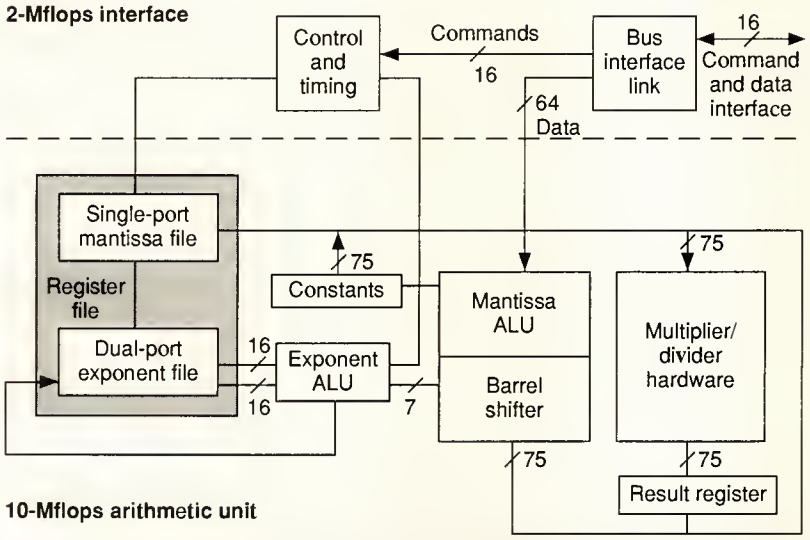
DSP processes 450 Mflops

The Desktop Signal Processor can connect to IBM PC/AT or Macintosh II hosts or run without a host in real-time applications involving control and monitoring functions. The multipurpose computer provides 25 to 450 Mflops when configured with one to 18 DSP32C processors. A full configuration provides aggregate data rates of up to 1,600 Mbytes/s.

With the processor, software developers can choose a C-like assembly language or AT&T's C and access a library of math, matrix, filter, and FFT functions. Additional features include a debugger and simulator and a DSIview library of subroutines for viewing and processing external signals through the I/O ports. **Dolphin Scientific; from $12,500; delivery 12 weeks ARO.**

Reader Service Number 17

Math processor aids laptops



The 16- or 20-MHz CMOS FasMath chip offers an extended, double-precision IEEE Std.-754 architecture with an 80-bit internal format for data storage and computation.

An 80386SX-compatible math processor promises up to five times the performance found in the Intel product. FasMath 83S87, designed for laptop and embedded controller markets, implements its floating-point primitive operations in hardware rather than in a microprogrammed sequencer. Three functional blocks—interface, execution, and control—make up the FasMath 83S87 architecture.

The execution unit performs floating-point primitive operations while the control unit supervises their execution, sequences primitives to realize complex operations, and controls traffic to and from the interface unit.

FasMath 83S87s can be used in desktop and laptop computing environments for image processing, circuit simulation, DSP, IC design and verification, and schematic and logic design applications. **Cyrix Corporation; $506 (16 MHz) and $556 (20 MHz) each.**

Reader Service Number 16

SPOX joins C30 development boards

The TMS320C30 development board, built around the 32-bit Texas Instruments floating-point processor, carries the SPOX real-time operating systems for DSPs. Useful for developing advanced video, instrumentation, and imaging systems, the SPOX/C30 system supplies 128 Kwords of 32-bit fast SRAM and 256-Kword, on-board memory capacity. **Spectrum Signal Processing Inc.**

Reader Service Number 18

Development tool plugs into ATs

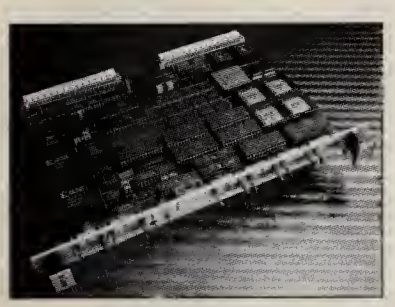
The DSP32C-AT lets users develop programs and hardware prototyping for systems incorporating the WE-DSP32C processor. The full-size IBM PC-AT or compatible board comes with 64 or 256 Kbytes of zero- or one-wait-state memory.

Also included in the plug-in board is a processor clock, dual analog I/O system,

and interface circuitry for the AT bus. Both channels of input and output operate simultaneously at a 50-kHz sample rate. A window-based debugger provides single stepping with automatic updates, breakpoints, program and data transfers, disassembly, and accumulator and register status. **DSP Design Tools; $2,395 (64 Kbytes), $2,995 (256 Kbytes).**

Reader Service Number 19

Speedy VMEbus/MAXbus board introduced



FDS recommends the VASP-16 to process DSP FIR filters, convolutions, correlations, and FFTs, as well as to evaluate development and final system solutions.

The VASP-16 integer digital signal processor combines a TMS320C25 scalar processor with four 30-MHz Zoran Vector coprocessors. VMEbus and multiple MAXbus 8- and 16-bit I/O ports allow input and output to run concurrently with processing. MAXbus lets the VASP-16 interface with compatible analog capture and storage boards for full DSP and image processing system solutions.

C Source, with a 90-function library, is available for PC, OS/9, and Sun platforms. **Future Digital Systems Ltd.**

Reader Service Number 20

Hardware, software RISCs

LR3000 meets military standard

Military temperature versions of the 16-MHz MIPS LR3000 CPU and LR3010 floating-point accelerator are currently being qualified for full Mil-Std-883C processing. Ceramic LCC and ceramic PGA production quantities exist now for use in temperatures from – 55°C to + 125°C. **LSI Logic; $187 (LR3000GX-16), $227 (LR3010GX-16) (100s).**

Reader Service Number 21

RISC responds in 100 µs

Series 8000, a MIPS R3000/3010-based family running the RTU 6.0 real-time Unix operating system, promises a 100-µs average response time. The series offers 20- to 160- MIPS computing engines featuring a multiprocessor architecture, fixed-priority task scheduling, and a choice of task-scheduling algorithms.

The three-system series consists of the 8300 and 8400 single or dual CPU and 8500 four-dual CPU configurations. Up to three additional dual CPU boards, memory, and a second VMEbus can be added to the 8500. **Concurrent Computer; from $55,900 (8300).**

Reader Service Number 22

Unix port added to 88000 CPU-8Xs

A Unix port for the Force Computers CPU-8X RISC family of 88000-based VME boards provides Unix capabilities for VME-based hardware. The port can be purchased in single-user object copies or in bulk licenses. **Opus Systems and Force Computers; $1,950.**

Reader Service Number 23

AIX ported to RS/6000

An IBM-endorsed company offers Version 3 of the AIX Unix variant for use on IBM RISC System/6000 workstations. AIX supports Berkeley 4.3, System V.3 APIs and commands, and proprietary Sunview tools. An additional capability helps users create distributed networks around the RS/6000. AGS.

Reader Service Number 24

R2000 aids telecommunications

A telephone switch, designed around the MIPS R2000, represents the telecommunications industry's first RISC-based product. The DMS-10 400E processor lets telephone companies using the DMS-10 family of digital central office switches offer rural customers a broader range of advanced services. Installed by removing 11 line cards and replacing them with three new processor cards, the 400E improves processing speed, expands memory, and increases the maximum line size from 10,800 to 12,000 lines. **MIPS Computer Systems.**

Reader Service Number 25

AMD RISC serves low-end applications

The Am29005, a software- and pin-compatible addition to the Am29000 RISC microprocessors, operates at 16 MHz for inclusion in 6- to 9-MIPS embedded systems.

Users can embed the 32-bit, three-bus, 9-MIPS microprocessor in cost-critical laser-beam printers and scanner systems. The 168-lead, plastic-quad-flat-pack processor supports the company's Fusion29K program for access to support and development software and hardware tools. **Advanced Micro Devices.**

Reader Service Number 26

Unix system combines LAN, graphics

A nonproprietary RISC system from Motorola offers Ethernet support, concurrent access to multiple applications, and high-resolution graphics. The Unix-based MultiPersonal Computer promises users performance up to 67.2 MIPS.

Designed for work groups, the three-

model system supports up to 66 color or black-and-white Network Display Stations per system. SoftPC, a software emulator of MS-DOS systems, and a trial copy of FrameMaker publishing software accompany MultiPersonal Computer. Each system includes one license and a copy of Uniplex II Plus integrated office automation software. **Motorola Computer Systems Division; from $23,985 plus $7,995 per-seat license.**

Reader Service Number 27

RS/6000s access scientific software

Version 4.3 of the Maple interactive computer algebra system will run on IBM RISC System/6000 products under the AIX V.3 operating system. Useful in scientific, engineering, and business math applications, Maple 4.3 contains libraries suitable for solving problems in robotics, chemistry, relativistic physics, structural mechanics, electrical engineering, and advanced mathematics. **Waterloo Maple Software; $3,045; academic discounts and site licenses available.**

Reader Service Number 28

Educate, develop programs with Simula

The 12.5-MIPS Sun Sparcstation now supports the Simula object-oriented programming language. Standard implementations include libraries for common problems in mathematical functions, random number generators, text manipulation, standardized file system interfaces, linked lists, and discrete event simulation. In addition, Simula provides interfaces to C, Fortran, and Pascal to facilitate use of libraries in these languages and to support easy migration to Simula and object-oriented programming.

A high-performance compiler in Simula aids program development and education. SIMDEB, a source-level, interactive debugger, lets programmers set breakpoints, examine and change variable values, follow pointers, display source code, and trace program execution. **Lund Software House AB.**

Reader Service Number 29

Ada available on RS/6000

Rational Environment, a software engineering Ada environment, has been integrated into RS/6000s running AIX 3.1. This integration produces a workstation-based development system compatible with products included in IBM's technical CASE solution.

An attached R1000 Series 300 coprocessor, a workstation interface software (called the X Interface to the RS/6000), and the Rational Environment form one system. The company also offers the Rational Design Facility, a Cadre Teamwork Interface, which integrates Teamwork on the RS/6000 with the Rational Environment, and an Interleaf TPS publishing interface. **Rational; $99,500 (system), $2,000 (Cadre), $1,000 (Interleaf).**

Reader Service Number 30

RS/6000s add mouse, icons

A version of the X.desktop user environment has been licensed to IBM as part of the AIXwindows Environment/6000 licensed program product. The AIXwindows Desktop package for RS/6000 Powerstations and Powerservers provides a graphical interface for day-to-day operations by hiding the complexities of the Unix operating system. Users can see icons, perform drag-and-drop actions, and manipulate a mouse for file management duties. **IXI Limited/IBM Corporation.**

Reader Service Number 31

Use CASE on RS/6000s

Users can access a series of CASE packages with RS/6000 AIX 3.1, PS/2, and RT systems. The TCAT family of test coverage analyzers provides fundamental support of software quality assurance by measuring the C1 coverage value for a test series. TCAT is available for Ada, C, Fortran, and Pascal.

CAPBAK captures keystroke sequences during programmer end-user testing of Unix application software and automatically plays back test sessions. The TDGEN test data generator system produces test data files in a user-designed format with choices of location-specified data, uniformly distributed random data, or value-factored data.

Additional packages include EXDIFF file comparison, SMARTS functionality verification, S-TCAT test completeness, and TCAT-PATH C1 coverage measurement tools. **Software Research; from $1,200 per product, depending on machine (size).**

Reader Service Number 32

RS/6000s gain RDBMS

An IBM/Ingres agreement lets RS/6000s run the Ingres relational database management system and access database data across AIX and SAA platforms. According to the company, applications port across the IBM workstation without recompilation, and applications created on an IBM RT port to the RS/6000. Ingres runs under the X Window System and supports the X.11 functionality of the RS/6000 and the X Station 120. **Ingres Corporation; from $5,000 per user.**

Reader Service Number 33

Model, animate 3D data on RS/6000

Advanced Visualizer on RS/6000 workstations lets 3D data be modeled, animated, and presented in photographic quality. Users can incorporate technical or scientific data into the system to produce accurate videos, films, and photos. The modeling package makes use of the RS/6000 floating-point processor, global superscalar processors, and optimized 3D graphics to let users manipulate data. **Wavefront Technologies;.$29,000 (software), $4,500 (1-year technical support); available 3Q.**

Reader Service Number 34

RISC VMEbus node processor introduced

A node processor, the V/FDDI 4211 Peregrine, attaches a VMEbus computer system or workstation to an FDDI network for communications protocol processing and network management. An Am29000 RISC processor executes 14 sustained MIPS in this application.

Dedicated execution memory for the Am29000 eliminates contention for mem-

ory access between the processor and the VMEbus or FDDI interfaces. A DMA BUSpacket interface decouples bus activity from other activity on the Peregrine through high-speed FIFO memory buffers and an asynchronous state machine that monitors VMEbus signalling. According to the company, BUSpacket tests transferred data over the VMEbus at 36 Mbytes/s.

The Peregrine can be configured to implement an FDDI single or dual attached station. **Interphase Corporation; $8,995 (single configuration), $10,995 (dual).**

Reader Service Number 37

Single-board computers

CAN chip conquers process control duties

The CP-526 computer based on the 80C535 microcontroller links to networks in general-purpose process control applications requiring high-speed serial communications. An 82526 Controller Area Network chip, originally designed to provide multiplexed communications for automotive in-vehicle control modules, provides the network link. Optional fiber optic interfaces and development software, including a system-monitor EPROM and cross assembler, also support the CP-526. **Allen Systems; $350 (standard configuration).**

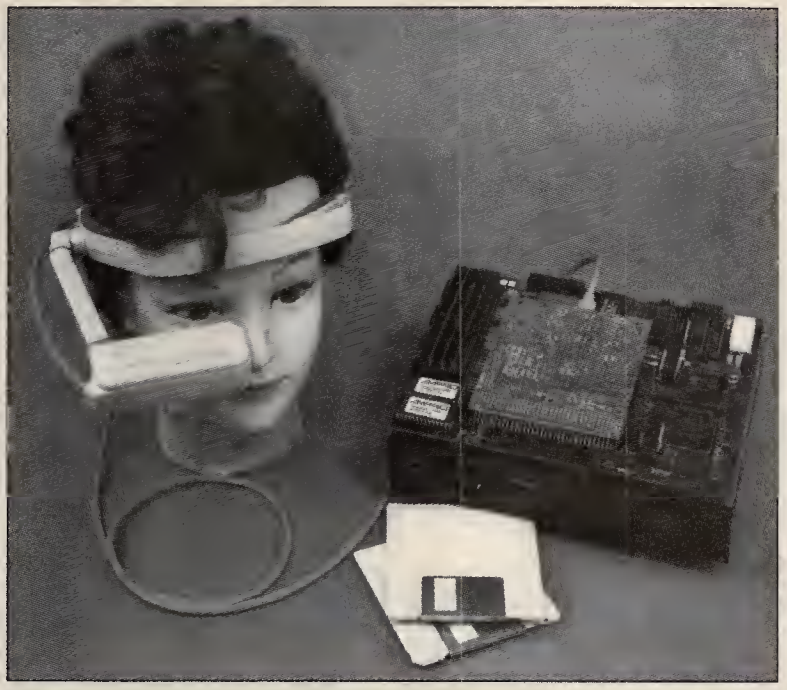
Reader Service Number 35

Eurocard delivers 4 MIPS

Surface-mount technology packs support hardware onto a 4-MIPS, STEbus Eurocard to ensure that systems designers can use the full capabilities of the 68020-based CPU board. The SC020T serves as a VMEbus alternative as well as an upgrade path for existing STEbus applications. A 0.5-Mbyte static RAM; 32-Kbyte, battery-backed static RAM; 0.5-Mbyte EPROM; and OS-9/68k multitasking operating system accompany the real-time SC020T. **Dean Microsystems; £745.**

Reader Service Number 36

Miniature images "float" before the eye



The Private Eye display can be powered by a Little Board to form a CPU, controller, and monitor the size of a 5.25-inch disk drive. An attached MiniModule circuit board interfaces with the system via PC-bus signals.

Users can translate a 12-inch PC image into a display they can view on a headset or hold in one hand. A MiniModule expansion board drives the two-ounce, 1.2 × 1.3 × 3.2-inch Private Eye display system.

Coupling the system with a 5.75 × 8 × 1.1-inch Little Board/386, /286, or /PC permits access to CGA-compatible graphics when used with PC/AT-based application software. Up to seven MiniModules can be supported in one Little Board-based system.

Private Eye uses + 5V power to display 720 × 280 pixels, which can be formatted as 25 lines with 80 characters per line or used to show bit-mapped graphics. **Ampro Computer; $250 (100s); delivery 45 days ARO.**

Reader Service Number 38

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 180 **Medium** 181 **High** 182

Product Summary

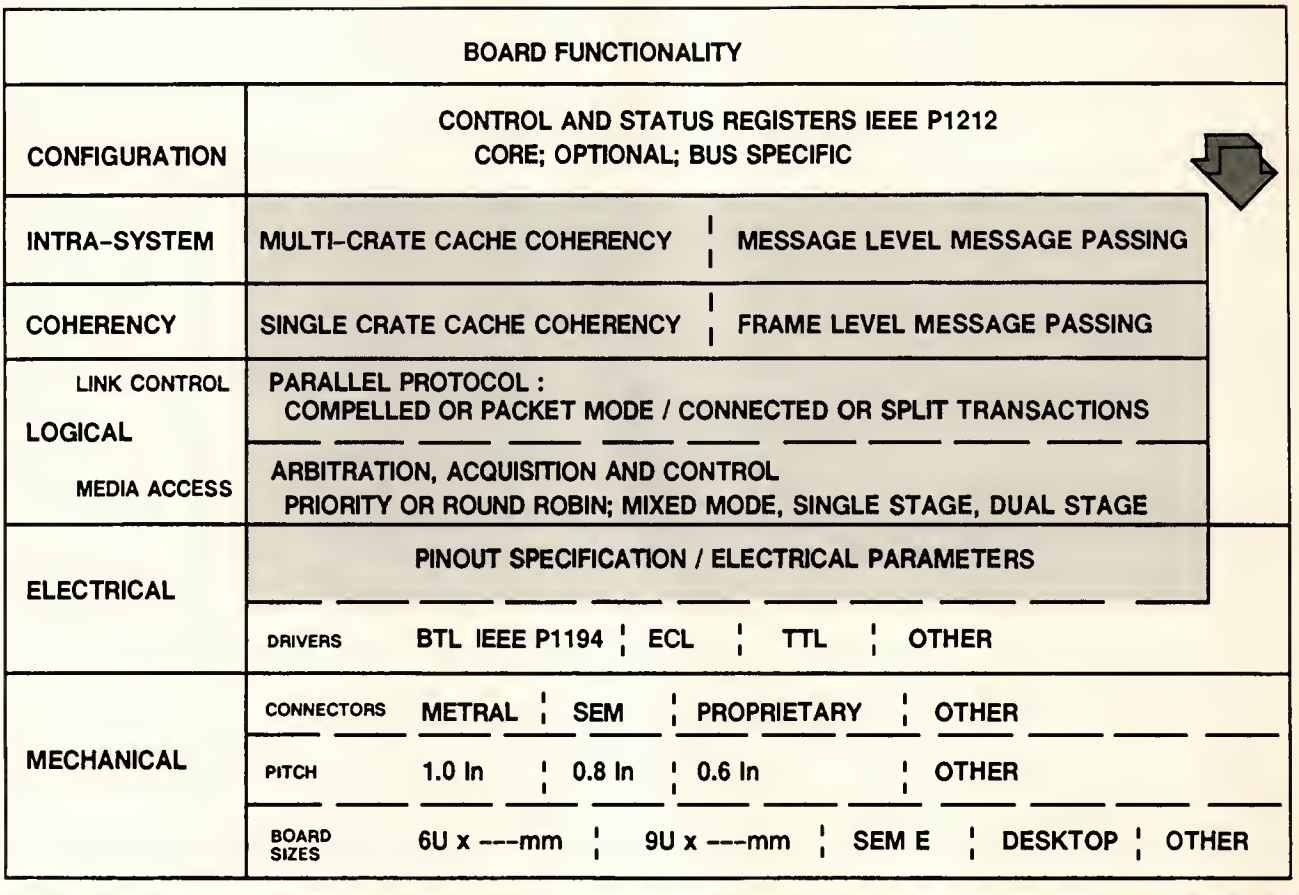
Marlin H. Mickle
University of Pittsburgh

For more information, circle the appropriate number on the Reader Service Card.

Manufacturer	Model	Comments	R.S. No.
Networks and communications			
Advanced Computer Communications	ACS 4000TR	Hardware option permits data between token-ring LANs to be transparently bridged or routed at aggregate speeds of up to 4 Mbps. The Series 4000 family member is configured into the ACS 4100/4400 hardware at time of order and also supports the Simple Network Management Protocol. $1,400.	80
Black Box	PC Modem Gateway	Cards link user PCs and synchronous mini- or mainframe environments and support V. 22 bis, V. 32 full-duplex, and half-duplex standards with 9,600-bps transmission. The software emulation packages include manual or automatic dial/answer support, a hot key for quick exit to DOS, and a user-redefinable keyboard. From $575 (card), $60 (software).	81
Exzel Corporation	Access Easy Network	Two Ethernet or Arcnet Novell network cards, cable, and software make up this small-business system. Two computers up to 200 feet apart can share information, software, and a printer. Under $100.	82
FTP Software	PC/TCP for OS/2	TCP/IP software lets OS/2-based PCs communicate with other PCs and computers with Unix, IBM VM, MVS, and VAX/VMS operating systems. The internetworking product includes a set of utilities and uses Presentation Manager for multitasking and windowing. US versions offer optional Kerberos security algorithm. $575.	83
General DataComm	MegaBridge	LAN-to-WAN system integrates multiple local and remote LANs into one communications network with protocol-transparent connectivity. The ISDN link's "learned address filtering" works to lower LAN congestion and improve traffic flow on the user's WAN. From $7,000.	84
KMW Systems	NetAxcess NuBus	AppleTalk-to-IBM midrange card plugs into Macintosh IIs to integrate IBM data; a Hypercard interface permits transparent database access. Users can use Laserwriters or Imagewriters as IBM System/3X printers. $3,995.	85

Manufacturer	Model	Comments	R.S. No.
Multi-Tech Systems	Multi-MuxX25 PAD	Packet assembler/disassembler comes in eight- or 16-port models that are field upgradable to full packet-switch capabilities. Both models meet X.3, X.28, and X.29 standards. RS232-C asynchronous channels operate to 19.2 Kbps; include autobaud; and connect to terminals, hosts, modems, or compatible devices. $3,999 (8 ports); $5,499 (16 ports).	86
Multi-Tech Systems	Multi-Com3780/BSC	EDI package includes a synchronous adapter card with RS232-C connector and communications software for use with IBM PC, XT, AT, or compatible computers. The half-size card accommodates 2400-, 4800-, or 9600-bps, external synchronous modems to communicate with remote mainframes or other BSC-capable devices. $899.	87
Network Software Associates	APPC Developer's Kit	Training, experimentation, and implementation package helps developers of LU6.2 APPC PC-to-host and LAN-to-host applications. Kit includes APPC connectivity software, interactive educational tools, APPC and SAA/CUA development aids, a file-transfer program, a performance monitor, technical support, and development assistance for one year. $995.	88
Network Software Associates	AdaptModem/2, V. 32	PS/2 MCA board provides built-in functionality for interactive, cooperative, and batch PS/2-to-host, LAN-to-host, PS/2-to-PC, and PS/2-to-PS/2 communications. One plug-in board combines a full-duplex synchronous modem, asynchronous modem, and SDLC adapter. AdaptSNA communications software supports 3270 emulation, cooperative processing LU6.2 APPC, batch-oriented 3770 remote job entry, and user-defined LU0. $1,295 (board), from $245 (software).	89
Standard Microsystems	COM90C66	Zero-wait-state, 16-bit Arcnet controller combines a PC/AT interface and on-chip, dual-port RAM. Software-compatible with the COM90C26 Arcnet controller, the three-chip set offers a Command Chaining feature that improves LAN throughput in high-traffic file-server applications. $24.50 (2,500s).	90
Tekelec	OSI Analyzer	Add-on tester for the Chameleon 32/20 data protocol test systems provides protocol developers and network managers with OSI test capabilities. Users can configure protocol stacks on existing company building blocks and customize or develop standard or hybrid protocol stacks. Developers can also link C protocols to the tester's building blocks to make customized stacks. From $17,000 (Chameleon test system with add-on).	91
Telebyte Technology	Model 63-2	Interface converter supports full-duplex transmission from an RS232 port while providing conversion circuitry to accommodate the differential I/O required by an RS422 port. The converter with programmable cable termination and status display supports a 38,400-bps data rate that can be transmitted over 4,000 feet of twisted-pair cables.	92
Triticom	ArcVision	Traffic-monitoring tool operates with Arcnet LANs and, when used with the company's adapter card, most PC/MS-DOS computers. Real-time display modes include 1) normal, to track packets sent by each station; 2) high speed, for extended performance; and 3) skyline, for aggregate traffic display over time. $150; $325 (adapter).	93

continued from p. 2



Futurebus+ profile options. The screened area indicates P896 documents.

be reserved for a 64-bit Futurebus+, although the 32 bits and supporting parity lines for A/D 32-63 may be left undriven. These pins are reserved to ensure interoperability of multiple bus width products. A 128-bit bus would logically extend into the next connector. The next figure depicts the bus arrangement for a 9U chassis configured to support a mix of 32-, 64-, and 128-bit products.

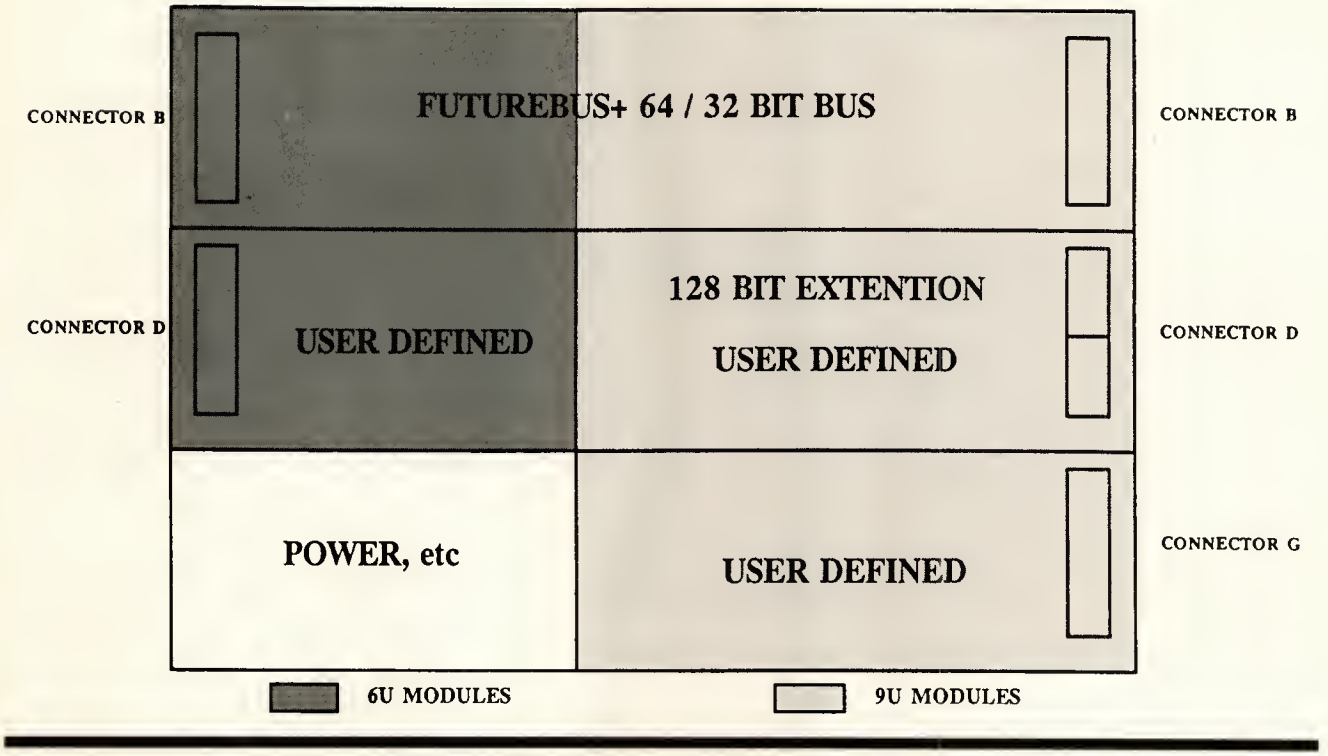
Futurebus+ media access is controlled by a functionally distributed, high-capability arbitration scheme. This protocol supports priority and round-robin arbitration in one or two acquisition cycles. A profile may choose the unrestricted or restricted arbitration mode. Idle bus arbitration (IBA) is available to support lightly loaded systems. Since all modules which adhere to the specification are required to support the unrestricted mode at initialization, all modules will interoperate at this level. The restricted mode and IBA are product enhancements. The system is configured through CSR registers, at initialization, to a common arbitration scheme.

Having become bus master via arbitration, logical link control is provided by the parallel protocol. Futurebus+ is a 64-bit architecture with a 32-bit subset and data extensions to 128 and 256 bits. The protocol supports connected and split transactions with compelled and packet-transfer modes. The handshaking lines provide flow control for both transaction types. Profile A mandates a 32-bit address and data width, and support for connected, compelled transactions.

Split and packet capability are optional, as is a 64-bit address/data path. CSR configuration registers are read at initialization to determine the mix of modules and capabilities and allow system parameters to be set. These optional features are product enhancements which increase overall performance but do not inhibit interoperability with less capable boards.

The Futurebus+ specification provides two system-level transaction paradigms: a tightly coupled shared-memory, cache-coherent scheme, and a loosely coupled message-passing scheme. It is left to the system integrator/profile developer to determine whether to mandate one or both methods. Communication between modules supporting opposite paradigms is possible via basic parallel protocol mechanisms. It is my belief that a standard (19-inch) chassis of the future will support several tightly coupled, cache-coherent compute engines (each consisting of a small number of modules) performing autonomous functions, that are loosely coupled via the message-passing paradigm.

The caching protocols provide snoop-



Futurebus + intermateability and interoperability.

ing, snarfing, and intervention with reflection. Snooping allows caching masters to monitor all transactions to the shared-memory space. Snarfing allows pertinent data to be removed from the bus transparently to update on-board caches, and intervention dictates that the owner of modified data intervene with a valid copy while the addressed slave "reflects" his drivers to update that cache line. It is expected that modules used in environments other than purely I/O buses will support the cache protocols.

The message-passing protocol standardizes at the frame and message levels, providing support for Futurebus+ messages or a number of other message schemes at the implementer's discretion. Message passing is easy to implement since it makes use of existing transaction types, uses a basic frame size compatible with cache lines, and is inherently a write-only method. Message passing enhances security features and supports real-time and fault-tolerant systems. While a given profile may mandate that a message may be implemented, Profile A does not require it. This added feature will allow product differentiation while not inhibiting interoperability. If implemented, the caching and message-passing protocols must be in accordance with the 896.1 specification.

Control and Status Registers reside at the top of the current Futurebus+ protocol stacks. This standard provides an organized register set for control, configuration, identification, and diagnostics between system bus nodes. CSR provides a software interface to Futurebus+ and other IEEE standards, in particular 1014.2 VME Futurebus+ Bridge, 1394 Serial Bus, and 1596 Scalable Coherent Interface. A profile may decide to specify implementation of the full register set or only bus-specific and core CSR registers.

A profile may address other issues such as board pitch, front-panel design, live insertion capability, environmental categories, system test, serial bus support, tag bit usage, connector keying, and byte ordering in a heterogeneous processor environment. Though it is not mandatory that these and other implementation issues be addressed, any items that require standard usage should be included in a profile. Proprietary profiles have the capability of being more restrictive than one developed in an open working group and may be used to meet more specific requirements.

Futurebus+ has justly been accused of being all things to all people. To fulfill the requirements of high-performance I/O bus usage, high-performance compute engines, and hierarchical multiple-bus systems is no easy task. The key element in operating at several points along this computing spectrum is a core set of specifications, defined by a profile, that provides several convergence layers. Each layer may have options to enhance the capabilities of that layer, but a profile specifies the minimum subset that guarantees interoperability.

While a profile option sounds at first like an oxymoron, this core set provides the convergence layer required for multivendor interoperability. Within Futurebus+, convergence is guaranteed at the connector footprint, pinout, bus location, unrestricted mode of arbitration, and connected, compelled parallel protocol. Module support for optional features provides product enhancement and differentiation, and system-level features.

Harrison Beasley
Fairfax, Virginia

FOR DISPLAY ADVERTISING INFORMATION, CONTACT:

Northern California and Pacific Northwest: Roy McDonald Assoc. Inc., 5915 Hollis St., Emeryville, CA 94608; (415) 653-2122.
Jim Olsen, P.O. Box 696, Hillsboro, OR 97123; (503) 640-2011.

Southern California and Mountain States: Richard C. Faust Co., 24050 Madison St., Suite 100, Torrance, CA 90505; (213) 373-9604.

Midwest: The Kingwill Company, 4433 W. Touhy Ave., Suite 540, Lincolnwood, IL 60091; (708) 675-5755.

East Coast: Atlantic Representative Group, 349 Maple Place, Keyport, NJ 07735; (201) 739-1444.

New England: Arpin Associates, P.O. Box 6444, Holliston, MA 01746; (508) 429-8907.

Europe: Heinz J. Görgens, Parkstrasse 8a, D-4054 Nettetal 1 - Hinsbeck (F.R.G.); phone: (0 21 53) 8 99 88; telex 841(17)2153310=HJG tlx d.

Southwest/Southeast: Heidi Rex, Office, 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1264; (714) 821-8380.

For production information, conference, and classified advertising, contact Heidi Rex or Marian Tibayan.

IEEE MICRO, 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1264; phone (714) 821-8380; fax (714) 821-4010.

continued from p. 96

constructs. RISC processors, on the other hand, provide the compiler with direct access to fundamental hardware functions and leave it up to the compiler to match these functions to high-level constructs.

Given this background, we can better evaluate what should be called RISC and what should not. Intel has added to the RISC confusion by claiming that the i486 uses RISC technology. In a panel discussion I moderated at Comdex last fall entitled "RISC and PCs: Is There a Marriage?," the Intel representative took the position: Yes, and that marriage is the 486. This sort of self-serving definition of RISC makes the term nearly devoid of meaning.

The i486 does indeed achieve many of the goals of RISC processors while executing a CISC instruction set. This fact does not, however, make it a RISC processor. The complexity of the i486 instruction set makes the processor's pipeline far more complex than that of any RISC machine. RISC is a characteristic of an architecture, not of an implementation; no processor that directly executes the i486 instruction set can possibly be a RISC machine. If you look at my lists of typical RISC characteristics, you'll see that the only one present in the i486 is the single-cycle execution of many instructions.

It is true that the i486 uses implementation techniques that are commonly employed in RISC processors. This approach doesn't make the i486 a RISC processor any more than putting a sports car engine and suspension in a sedan makes that sedan a sports car. The sedan might achieve performance close to that of a sports car, but only a marketeer would call it a sports car.

IBM's RS/6000 has also raised questions about what defines a RISC processor. How, say the skeptics, can you call it a RISC when it has string move instructions and complex instructions such as multiply-and-add? I believe that the RS/6000 does qualify as a RISC processor because it adheres to the philosophy stated in my definition and also exhibits nearly all of the characteristics listed here.

Perhaps the most extreme abuse of the term *RISC* occurred when Microchip Technology introduced a CMOS version of the old PIC 8-bit microcontroller, calling it a RISC microcontroller. The press, for the most part, picked this term up from the product release and repeated it without questioning its validity. While the instruction set is indeed small and many of the instructions execute in one clock cycle, the controller does not include any of the other characteristics listed here. More fundamentally, it fails to meet either of the criteria in my definition. The processor does not support high-level languages (so it cannot have been designed for code generation by a compiler), and the processor is not pipelined.

Various redefinitions of RISC have been proposed, including *reduced instruction-set complexity* and *reduced instruction-set cycles*. These terms only further confuse the issue, however, and perhaps it is best to forget that RISC is an acronym at all, just as RAM no longer means random access memory.

Why belabor this point? When words are abused—either to further marketing aims or simply because the words are poorly understood—they begin to lose meaning. More importantly, true RISC processors offer significant advantages that cannot be gained by using CISC architectures. Because RISC machines are simpler to implement, they can be implemented in a wider range of technologies, in a shorter period of time, and with fewer errors than the CISC variety.

Perhaps most importantly, superscalar implementations of true RISC architectures will be easier to produce. If you consider the problem of decoding multiple instructions in parallel, the problem clearly becomes far more difficult when the instructions have variable lengths. Thus, the performance gap between true RISC and CISC processors is likely to widen as superscalar RISC implementations become commonplace.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Service Card.

Low 186 **Medium** 187 **High** 188

Department

Micro View

Michael Slater
Microprocessor Report
(707) 823-4004
mslater@cup.portal.com

What is RISC?

The term *RISC* has become one of the most widely used—and abused—acronyms in computer architecture. Various companies have distorted the term to meet their marketing objectives. Countless observers who lack a full understanding of the origins and intent of RISC architectures remain understandably confused. In this column, I explain the origins of the term, some of its misuses, and my preferred definition.

The name itself—reduced instruction-set computing—lies at the root of the confusion. David Patterson, who led the original RISC project at the University of California, Berkeley, coined this name. The Berkeley RISC machine does indeed have a very reduced instruction set, and RISC is a nice, catchy name. However, if you use the number of instructions as the sole determining factor, devices such as the PDP 8 and the 6502 also qualify as RISC machines.

A variety of constraints, such as the need to complete projects within the academic calendar, influenced the simplicity of the Berkeley RISC design. Thus it is not reasonable to use the features of this design as the defining characteristics of RISC.

No official definition of RISC exists. In fact, even unofficial definitions are hard to come by. RISC is neither an architecture nor a technology. It is a philosophy of computer design.

The term *RISC* commonly refers to a loosely defined class of architectures that share a number of common features.

For the term to be useful, its definition should encompass all of the architectures that are widely accepted as RISC, while excluding those that are not. Although no definition is perfect, I propose the following:

> A RISC processor has an instruction set that is designed for efficient execution by a pipelined processor and for code generation by an optimizing compiler.

While this definition doesn't include any specific features and is open to broad interpretation, I believe that it sums up the central philosophy of RISC designers. All of the characteristics associated with RISC processors derive from this philosophy. The characteristics that derive from the desire for efficient execution by a pipelined processor include

- fixed-length instructions;
- simple, consistent instruction encoding;
- single-cycle execution of most instructions;
- a memory accessed only by load and store instructions;
- simple addressing modes;
- delayed branches; and
- load delay slots.

Compiler considerations constitute an equally important part of the RISC philosophy. In fact, one partially sarcastic definition of RISC is *relegate the impossible stuff to compilers*. Compiler considerations influence a number of features, including

- a three-address instruction format,
- a large register set, and
- a symmetrical instruction set.

Note that none of these features rule out instructions that perform complex functions. Floating-point calculations can indeed properly exist within the RISC framework. RISC processors include such functions only if they are significantly faster when implemented in hardware rather than as a sequence of simple instructions. Complex addressing modes serve as one example of a function that fails this test. The word *reduced* does not imply reduced to a bare minimum; it means reduced to those instructions that are justified by performance gains.

Another aspect of the RISC philosophy is that the compiler should be exposed to the basic capabilities of the machine. CISC processors hide these basic capabilities, stringing together basic functions with microcode to form machine language instructions. CISC processors hide the details of the processor's hardware from the programmer to provide a more elegant machine language that better matches high-level

continued on p. 95

0272-1732/90/0600-0096$01.00 © 1990 IEEE

IEEE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Computer Society Offices

Headquarters Office
1730 Massachusetts Ave. NW
Washington, DC 20036-1903
Phone (202) 371-0101
Fax: (202)728-9614

Publications Office
10662 Los Vaqueros Cir.
PO Box 3014
Los Alamitos, CA 90720-1264
Membership and General Information: (714) 821-8380
Publication Orders: (800) 272-6657
Fax: (714) 821-4010

European Office
13, Ave. de L'Aquilon
B-1200 Brussels, Belgium
Phone: 32 (2) 770-21-98
Fax: 32 (2) 770-85-05

Asian Office
Ooshima Building
2-19-1 Minami-Aoyama, Minato-ku
Tokyo 107, Japan
Phone: 81 (3) 408-3118
Fax: 81 (3) 408-3553

Use the Reader Service Card to obtain information on:

- Membership application—student #203, others #202
- Perodicals subscription form for individuals #200
- Periodicals subscription form for organizations #199
- Publications catalog #201
- Standards status report #195
- Compmail+ international electronic mail/database brochure #194
- Technical committee list/application #197
- Chapters lists, start-up procedures—student/regular #193
- Student scholarship information #192
- Volunteer leaders/staff directory #196
- IEEE senior member application #204

To check membership status or report a change of address, call the IEEE toll-free number, 1-800-678-4333. Direct all other Computer Society related questions to the Publications Office.

Purpose

The IEEE Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 100,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

Members receive the acclaimed monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

Computer. An authoritative, easy-to-read magazine containing tutorial and in-depth articles on topics across the computer field, plus news, conferences, calendar, interviews, and new products.

Periodicals. The society publishes six magazines and five research transactions. Refer to membership application or request information as noted above.

Conference Proceedings, Tutorial Texts, Standard Documents. The Computer Society Press publishes more than 100 titles every year.

Standards Working Groups. Over 100 of these groups produce IEEE standards used throughout the industrial world.

Technical Committees. More than 30 TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.

Conferences/Education. The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.

Chapters. Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

European Office

Payments for Computer Society membership and publication orders are accepted by checks in Belgian, British, German, Swiss, or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by American Express, MasterCard, or Visa credit cards.

Asian Office

Payments for Computer Society membership and publication orders are accepted by checks in Japanese or US currency. Checks in US funds must be drawn on a US bank. Payment may also be made by electronic fund transfer to the Bank of Tokyo, Akasaka Branch, Toza acct. 0767956; the credit receiver is the IEEE Computer Society Headquarters Office. Payment may also be made by American Express, MasterCard, or Visa credit cards.

Ombudsman

Members experiencing problems — magazine delivery, membership status, or unresolved complaints — may write to the ombudsman at the Publications Office.